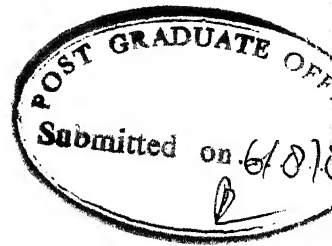# SYNTHESIS OF STATIC CONTEXT IN PASCAL PROGRAMS WITH L-ATTRIBUTE GRAMMARS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By

PUKHRAJ KACHHWAHA

to the
DEPARTMENT OF COMPUTER SCIENCE
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
AUGUST, 1981

CERTIFICATE

This is to certify that the thesis entitled "SYNTHESIS OF STATIC CONTEXT IN PASCAL PROGRAMS WITH L-ATTRIBUTE GRAMMARS" has been carried out by Sri Pukhraj Kachhwaha under my supervision and has not been submitted elsewhere for the award of a degree.

Kanpur
July 1981

Kesav V. Nori
Assistant Professor
Computer Science Programme
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

CS- 1981- M- KAC- SYN

CONTENTS

ACKNOWLEDGEMENTS

At the onset, I wish to express my heartfelt gratitude to Professor K.V. Nori for his able guidance, keen interest and continued encouragement in the initiation, progress and conclusion of this project. It was a pleasant experience to work under his expert guidance, enjoying his friendly cooperation.

Special reference should be made of Mr. Karnal Singh. He has closely worked with me throughout the course of work. I cherish enduring memories of umpteen pleasant sessions of long, fruitful and stimulating discussions with him.

Friends, especially, Datta, Bhatta, Sridhar and Mao, were of significant help in keeping my spirits high with pleasant company thus making my stay of two years at IIT Kanpur thoroughly enjoyable and worthwhile.

Special thanks are due to Mr. H.K. Nathani for his excellent typing.

Kanpur
July 1981

— Pukhraj Kachhwaha

ABSTRACT

This thesis is concerned with the synthesis of static context in Pascal programs. This context is synthesized during the declaration part of a program, to be used in performing context sensitive analysis during its action part. L-Extended Attribute Grammars (L-EAG) are used as a formal specification mechanism to build the static context. We have derived an L-EAG for PASCAL, conforming to the ISO standard. We describe the structuring techniques and operators of the abstract data types, which define and synthesize attribute domains of the L-EAG. Various domains used in synthesizing the context are presented illustrating their general structure. The transformation rules for interpreting an attribute grammar specification as a program are derived. This brings us closer towards realising an automatic context sensitive analyser generator system. The transformation rules are adapted to hand code a program which synthesizes the static context in Pascal programs.

CHAPTER I

PROLOGUE

1.1 MOTIVATION

We are interested in systematic development of context sensitive analysers for programming languages. To this end, we must show that it correctly accepts or rejects an input string, based on the specification of the semantics of the language.

The strong formal background of context-free grammers (CFGs) for exact syntactic specification of a programming language makes automatic generation of parsers an easy task. However, inspite of the presence of various mechanisms for formal specification of context sensitive aspects of a programming language, none of them is strong enough to produce an automatic Context-Sensitive Analyzer generator system – since the transformation rules for conversion from formal specification to programs are still not well established. We obtain a formal specification for the different context sensitive issues of a programming language in a manner suitable for ready implementation. The formal specification facilitates the proof of correctness of the developed program since the validity of the basic transformation rules ensure the correctness of the

1

entire program. The formal specification, even, reveals hidden semantic irregularities of a programming language and thus helps in correct implementation of the defined language.

1.2 INTRODUCTION

We have chosen the programming language standard PASCAL, ISO defined [Add 80], for the systematic development of Context Sensitive Analysis (CSA) phase of its compiler. We assume availability of a context free grammar (CFG) specifying syntax of PASCAL.

CFGs can only determine the context-free validity of an input string but they fail to ascertain the validity of an input string under the semantic rules defined by the language. Specifically CFGs cannot tackle the problem of scope rules, declaration before use, consistency of use with respect to declaration and detection of improper uses in a programming language like PASCAL. Thus we need to have a formal specification mechanism capable of specifying context sensitive issues of a programming language.

Although not capable of specifying context sensitive issues of a programming language, CFGs possess the desirable properties of readability, strong formalism and the deterministic CFGs used in practice ensure

unambiguity of language definition. Thus in all likli-
hood, more powerful mechanism for formal specification
of CSA that matches the success of CFGs will have to be
a clean extension of CFG.

A few well known formalisms available are van
Wijngaarden grammars, Vienna Definition Languages and
Attribute Grammars [MLB 77]. We have chosen L-Extended
Attribute grammar specification mechanism (an enhanced
version of attribute grammars) mainly due to the ease
with which it can be integrated with LL-1 parsing.
Attribute grammars offer the advantagesof readability
because of their explicit attribute structure and
distinction between "inherited" and "synthesized"
attributes and the clear visibility of the underlying
context free syntax. This also makes attribute grammars
suitable for automatic compiler construction. The other
formalisms tend to produce language definition which are
not easily readable and do not offer the advantages
inherently present in attribute grammars.

1.3 SCOPE OF THESIS

This thesis deals with the synthesis of static
context in PASCAL programs. Static context synthesized
during declaration part of the Pascal programs will be
required for performing Context sensitive analysis during
their action part.

Each identifier declared in Pascal programs is associated with a set of attributes. Upon occurrence of declaration of an identifier in Pascal program, this set of attributes are evaluated and the previous context is updated by inserting in it the name of the identifier alongwith its synthesized attributes.

Structure of declaration and semantics rule of the programming language dictate the rules for attribute evaluation of a declared object and determining exactly at which place this object alongwith its synthesized attributes is to be put in the context.

Example

If an identifier is declared in the CONST declaration of Pascal program then it is put in the environment of the Block at top of the SYMBOL TABLE, whereas if the identifier is declared as a field of the record it is put in the local environment of the RECORD structure. Further if the field declared in a record is of enumeration type then the semantics rules of PASCAL dictate that the corresponding enumerated constants encountered are to be put in the environment of the Block at top of the SYMBOL TABLE.

1.4 STRUCTURE OF THESIS

Chapter II deals with the L-Attribute grammars and
their use in specification of static context. Section
2.1 is a brief presentation of attribute grammars, their
definition and fundamental problems associated with them.
Next section describes L-Attribute grammars, nature of
dependencies in them and their suitability of integration
with LL-1 parsers. Finally the use of L-EAG for
synthesis of static context is illustrated. We discuss
about various domains, abstract data types and the
operators associated. Local extensions to EAG are also
described.

Chapter III discusses the implementation of static
Context Synthesizer. Transformation rules to convert
formal specification in attribute grammar to a Pascal
program and Pragmatic Transformation Strategy (PTS) are
derived in first section. These are, subsequently,
illustrated through an example of generating program
segment from L-EAG production. Next we describe
various significant domains and their structure used
in the Symbol Table construction. In the end, we
briefly present our experiences in program construction
by transformation rules.

The concluding Chapter IV describes some observations and experiences regarding transformation rules, hand coding of the routine for the CSA phase of the PASCAL compiler and the utility of a formal specification of the problem. It also points out possible improvements in program coding and provides some suggestions.

—

CHAPTER II

L-ATTRIBUTE GRAMMARS AND THEIR USE IN SPECIFICATION OF STATIC CONTEXT

2.1 ATTRIBUTE GRAMMARS

2.1.1 DEFINITION

An attribute grammar is an ordinary context free grammar augmented with attributes and semantic functions and constraints.

Formally it can be defined as [WM 77]

$$G = <D, V, \tilde{S}, B, P>$$

whose elements are defined in the following paragraphs. $D = (D_1, D_2, \ldots, f_1, f_2, \ldots)$ is an algebraic structure with domains $D_1, D_2, \ldots$, and (partial) functions $f_1, f_2, \ldots$ operating on cartesian products of these domains. Each object in one of these domains is called an <u>attribute</u>. V is the vocabulary of G. $V = V_N \cup V_T$ where V_N = Non-terminal symbols and V_T = Terminal symbols. Each symbol in V is associated with a fixed number of <u>attribute positions</u>. Each attribute position has a fixed domain from D, and is classified as either "<u>inherited</u>" or "<u>synthesized</u>".

The attributes of a symbol $X \in V$ identify the various components of its "meaning". Inherited attributes transmit information down the parse tree toward the leaves, while synthezied attributes transmit information upto the tree toward the root.

S, a member of V_N, is distinguished nonterminal or start symbol of G.

Start symbol has no attribute positions since it can have no ancestors. Terminal symbols have no inherited attribute positions as they have no associated semantics.

Let v be any symbol in V having $\underline{p}$ attribute positions, whose domains are D_1, ... D_p respectively. If a_1, a_2, ..., a_p are attribute variables in the domains D_1, ..., D_p respectively, then

v ↕ a_1 ... ↕ a_p is an <u>attributed symbol</u>. Where each ↕ denotes either ↓ or ↑, prefixing an inherited or synthesized attribute position, as the case may be.
B is a finite collection of attribute variables. Each variable has a fixed domain chosen from D.
P is a finite set of <u>production rule forms</u>, each of the following form:

p: $X_0 \rightarrow X_1 X_2 ... X_{np}$ and a set of <u>semantic rules</u>.
where X_0, X_1, X_2, ..., X_{np} are attributed symbols.
If an attribute a belongs to attributes associated with $\underline{X_k}$ then $(\underline{a,k})$ is defined as an <u>attribute occurence</u> in production $\underline{p}$.

Inherited attribute positions on LHS and synthesized attribute positions on RHS of a production rule are called <u>defining positions</u>. Synthesized attribute positions on

LHS and inherited attribute positions on RHS of a production rule are called <u>applied positions</u>. This classification is illustrated below:

$$V + \overline{\ldots} + \overline{\ldots} \quad \rightarrow V_1 + \overline{\ldots} + \overline{\ldots} , \overline{\ldots}, V_{np} + \overline{\ldots} + \overline{\ldots}$$

$$\text{def} \quad \text{app} \qquad \text{app} \quad \text{def} \qquad\qquad \text{app} \quad \text{def}$$

<u>Semantic rules</u> consist of <u>semantic functions</u> and <u>constraints</u>

<u>Semantic functions</u>

For each attribute occurrence (a,k) at an applied position in production p, there is an associated semantic function $f^p_{(a,k)}$. The semantic function determines the value of the attribute occurrence as a function of values of certain other attribute occurrences in the same production. It specifies the meaning of a parse tree locally, in terms of only a node and its immediate descendents. The dependence of an attribute evaluation on other attributes gives rise to the concept of the dependency set [Boc 76] which is defined as follows:

Dependency set $D^p_{(a,k)}$ is the set of attribute occurrences whose values are used for the evaluation of the attribute occurrence (a,k) by semantic function $f^p_{(a,k)}$. Thus the semantic function $f^p_{(a,k)}$ cannot be evaluated until all attribute occurrences in its dependency set $D^p_{(a,k)}$ have been computed. This constraint is known as <u>Dependency Constraint</u>.

Constraints

They are expressed as a relation between attribute variables and must be satisfied in each application of the associated production rule.

2.1.2 FUNDAMENTAL PROBLEMS

(1) <u>Circularity</u>: It is concerned with the dependency of evaluation of two attributes on one another and is defined in terms of Dependency Graph [KR 79].

<u>Dependency Graph</u>: Dependency graph for a production p, denoted by DG_p, is a tuple $(D\ V_p,\ DE_p)$. Where DV_p is a set of vertices each of which is an attribute occurrence in production p and DE_p is the set of directed edges joining two vertices V_{p1} and V_{p2}.

Formally it is defined as:

$$DG_p = (DV_p,\ DE_p)$$

where

$$DV_p = \{(a,k) \mid a \in A(X_k),\ 0 < = K < = nP \}$$

and $A(X_k)$ = attribute occurrences associated with X_k

and

$$DE_p = \{<(a_1,k_1),\ (a_2,k_2)> \mid (a_1,k_1) \in D^p_{(a_2,k_2)}\}$$

There is a directed arc from attribute occurrence O_1 to occurrence O_2 if O_1 belongs to dependency set of O_2.

Dependency Graph (DG) for a parse tree of a sentence in the attribute grammar is obtained by pasting together DG_p for each production occurring in the parse tree.

Now we can define circularity in terms of Dependency Graphs. If the Dependency Graph of a parse tree contains a directed cycle, then some of its attributes cannot be determined as they depend on one another. Such a tree is semantically circular and the corresponding attribute grammar is also circular.

(2) <u>Order of Evaluation of Attributes</u>: Order of evaluation of attributes in a parse tree must satisfy the dependency constraints.

Thus the use of an attribute grammar requres to show that there are no circularities present in it. Testing for circularity has been described in [Knu 68]. Dependency constraint is the only constraint on the order of evaluation of attributes imposed by an attribute grammar. Thus we must find a specific order of evaluation of attributes in an attribute grammar production which obeys this constraint.

2.2 <u>L-ATTRIBUTE GRAMMARS</u>

Consider an attribute grammar rule of the following form:

$$V \downarrow \dots \uparrow \dots \rightarrow V_1 \downarrow \dots \uparrow \dots, \dots, V_m \downarrow \dots \uparrow \dots$$

An attribute grammar is L-attributed, if and only if, in every such rule, and for every $i = 1, \ldots, m$, no inherited attribute of V_i depends on a synthesized attribute of any of $V_i, \ldots, V_m$.

Given an attribute grammar, the attributes of any derivation tree can be evaluated from left to right if dependency sets of semantic functions of any production $p: X_0 \to X_1 \ldots X_{np}$ of the grammar satisfy the conditions:

(1) $D^p_{(s,o)} \cap S_o = \emptyset$

For all synthesized attributes $s \in S(X_o)$; where $S_o = S(X_o)$

(2) $D^p_{(i,k)} \cap S_o \cup^{np}_{j=k}(I_j \cup S_j) = \emptyset$

For all $K = 1, \ldots, n_p$ and $i \in I(X_k)$; Here I stands for inherited and S for synthesized attributes.

L-attribute grammars satisfy these conditions. Hence, if the underlying CFG of an L-attribute Grammar is LL(1), attribute evaluation can be performed along-with the parse tree construction in a single left to right pass.

The evaluation of attributes in an L-AG production of form (1) proceeds as:

For each V_i, $i = 1, \ldots, m$, first its inherited attributes are evaluated (using the inherited attributes of $\underline{v}$ and the synthesized attributes of $V_1, \ldots, V_{i-1}$),

then the subtree under $\underline{v}_i$ is traversed, resulting in
the evaluation of the synthesized attributes of $\underline{v}_i$.
Finally any constraints are checked, and the synthe-
sized attributes of $\underline{v}$ are evaluated (using all the
attributes in the defining positions of this production),
thus completing the traverse of the subtree under $\underline{v}$.

2-3 L-EXTENDED ATTRIBUTE GRAMMARS

The essential difference between an attribute
grammar and an extended attribute grammar is that the
attribute positions in an EAG production rule may be
occupied by attribute expressions rather than just
attribute variables. This relaxation allows all rela-
tionships among the attributes in each productions to be
expressed implicitly so that explicit semantic rules and
constraints become unnecessary.

Formally:

A production of EAG is:

$p\colon X_0 \dashrightarrow X_1 \dots X_{np}$ where X_i for $i = 0, \dots, n_p$
is an attributed symbol represented as:

$$X_i \updownarrow e_1 \updownarrow e_2 \dots \updownarrow e_k$$

where e_i is an attribute expression.

An attribute expression is one of the following:

(1) A constant attribute

(2) An attribute variable

(3) $f(e_1, \ldots, e_m)$ where $e_1, \ldots, e_m$ are attribute
expressions and f is an appropriate (partial)
function chosen from domain D.

EAGs are intended to preserve all the desirable proper-
ties of attribute grammars, but at the same time are more
concise and readable. They allow generative definition
of languages rather than algorithmic definition.

L-attribute grammar with the same augmented rules
is called L-extended Attribute Grammar.

2.4 USE OF L-EAG FOR STATIC CONTEXT

We have specified all the context sensitive issues
of PASCAL in L-EAG. [Wat 79] This gives a formal specification
of the static context in Pascal programs. This formalism
dictates the algorithm for synthesizing the context in
Pascal programs. In this section we describe abstract
data types, alongwith their operators, used in L-EAG.
Local extension to EAG and a few deviation from standard
EAG are also discussed. These modification preserve
the basic properties of L-EAG.

2.4.1 ATTRIBUTE DOMAINS AND THEIR OPERATORS

All the attribute domains used in L-EAG specification
are listed in Appendix 1. This list consists of base

domains as well as of complex domains. Complex domains are constructed from the base domains using composition rules employing the concept of abstract data types [DDH 72]. We, briefly, present various abstract data types alongwith their operators below:

1. SET

Let D be a domain and S = {D}. Then $\underline{S}$ is a domain consisting of values from powers of D. If $x \in S$ then $x \in 2^D \Rightarrow x \subseteq D$. S is defined as a set. Various operators defined over such sets are:

(a) Membership, (b) Union (c) Set difference (in our case this is used to delete an element from the set).

The meaning of these operators is self-explanatory.

2. CARTESIAN PRODUCT

If T_1, ..., T_n are domains and g_1, ..., g_n are distinct names, then $P = (g_1 : T_1; ... ; g_n : T_n)$ is a cartesian product with field selectors g_1, ..., g_n. For every a_1, in T_1, ..., every a_n in T_n, $(a_1, a_2, ... a_n)$ is in P. This is the composition function for the cartesian. Operator defined over cartesian product is Selection: for every p in P and for every $i = 1$, ..., n, $P . q_i$ is in T_i and denotes the ith filled of P.

Example:

MODE = (kind : KIND; type : TYPE; level : LEVEL);
MODE is composed of three component domains KIND, TYPE
and LEVEL. MODE kind denotes the domain KIND.

3. DISCRIMINATED UNION

If $T_1, \ldots, T_n$ are domains (or cartesion products
of domains), $g_1, \ldots, g_n$ and $t_1, \ldots, t_n$ are distinct
names, then

$$U = (SEL : SELECTOR; (g_1(t_1 \cdot T_1) \mid g_2(t_2 : T_2) \mid \ldots$$
$$\mid n(t_n : T_n))$$

is a discriminated union with selectors $g_1, \ldots, g$.
$t, \ldots, t_n$ are distinct names. For every $i = 1, \ldots, n$,
g_i is a composition function from T_i to U. If any T_i
is void, then we abbreviate $g_i(t_i : T_i)$ to g_i. SELECTOR
is a domain consisting of composition functions $g_1, \ldots,$
g_n and SEL is in SELECTOR. At a time only one composi-
tion function g_i is active which defines U. SEL acts as
a discriminator to decide which g_i is active.

Operator defined on Discriminated union U is selec-
tion:

If $SEL = g_i$ then T_i defines U and $U.t_i$ is accessible.

4. MAPS

If D and R are domains, then $M = \{ D \rightarrow R \}$ is the
domain of (partial maps) from D to R. { } denotes the
map defined at no point in D. Let $d_1, \ldots, d_n$ be
distinct elements of D and $r_1, \ldots, r_n$ be in R.

If $m = \{d_1 \to r_1, \ldots, d_n \to r_n\}$ and m is in M then m denotes a map defined at points $d_1, d_2, \ldots, d_n$ and nowhere else.

Operators defined on Maps

(a) <u>Membership</u>: Membership of d in D is defined if $(d \to r) \in$ where r is in R and m is in M.

(b) <u>Address</u>: If membership of d is defined in m then address(d) gives the address of entry $(d \to r)$ in map m.

(c) <u>APPLY</u>: For every d in D and m in M, m[d] either is in R (i.e., $(d \to r) \in m$, where $r \in R$) or is undefined. This is also known as the application function for a map M.

(d) <u>Disjoint Union</u>: For each m_1 and m_2 in M, $M_1 \emptyset m_2$ is the disjoint union of m_1 and m_2. $m_1 \emptyset m_2$ is undefined if, for any d in D, both $m_1[d]$ and $m_2[d]$ are defined, otherwise

$(m_1 \emptyset m_2) [d] \equiv$ if $m_1[d]$ is defined then $m_1[d]$

else $m_2[d]$

(e) <u>Overridden</u>: For each m_1 and m_2 in M, m_1/m_2 is the map m_1 overridden by map m_2; i.e.,

$(m_1/m_2)[d] =$ if $m_2[d]$ is undefined then $m_1[d]$ else $m_2[d]$

(f) <u>Modify</u>: Let $(d \to r) \in m$ where $d \in D$ and $r \in R$. Consider a map $d \to r_1$, $r_1 \in R$. modify $(d \to r)$ to $(d \to r_1)$ in m is accomplished as follows: Delete the element

$(d \rightarrow r)$ from domain $m \in M$ and then take disjoint union of m with $\{d \rightarrow r_1\}$ i.e., $m \not{\cup} \{d \rightarrow r_1\}$.

Example

$$ENVIRON = \{ NAME \rightarrow MODE \}$$

Let Symbol table be an attribute of domain ENVIRON. APPLY function applied to symbol table and name indicates whether an identifier is previously declared, and if declared, it returns the attributes of the identifier (i.e., MODE of identifier).

Disjoint Union helps in detecting multiple declarations.

Overriden rule is useful for implementing scope rules of a block structured language.

5. SEQUENCES

If D is a domain then $S = D^*$ is the domain of sequences of elements of D [ ] denotes an empty sequence. If s is in S and d D, then d^-s denotes the sequence obtained by appending $\underline{d}$ to the left of $\underline{s}$.

Operators defined on sequences

(a) Occurrence test: It checks whether d is in s or not.

(b) Appendlast: Appendlast $(S,d) = s^-d$

(c) Appendfirst: Append first $(s,d) = d^-s$

(d) delete(d): Delete all occurrences of $\underline{d}$ in $\underline{s}$.

(e) disjoint append: If occurrence test (s,d) = false
then Appendlast (s,d) else undefined.

(f) <u>CAR</u>: CAR(s) gives the first element of s.

(g) <u>CDR</u>: CDR(s) gives the remaining sequence after deleting the first element.

2.4.2 <u>LOCAL EXTENSIONS TO EAG</u>

In general, an EAG is a straight forward extension of context free grammar. However, in our application, it is convenient to use Right Regular Part (RRP) CFGs. Therefore the EAGs we need require to be suitably extended to provide for the occurrence of the Kleene operator ($\{\ \}^*$) in the right hand side of the productions. The extension to EAG is illustrated through an example.

Consider a production with Kleene operator in the RHS expressed in extended EAG form:

$$\langle Term \downarrow x \uparrow x_2 \rangle \rightarrow \langle Factor \downarrow x \uparrow x_1 \rangle \uparrow x_1 \{\langle MULOP \rangle$$

$$\langle Factor \downarrow x_1 \uparrow x_3 \}^* \uparrow x_3 \ \langle Here \ x_2 = x_1 \rangle$$

(1)

This production is expressed in standard EAG with an underlying conventional CFG as:

$$\langle Term \downarrow x \uparrow x_2 \rangle \rightarrow \langle Factor \downarrow x \uparrow x_1 \rangle \ \langle A \downarrow x_1 \uparrow x_2 \rangle \quad (2)$$

$$\langle A \downarrow x_1 \uparrow x_2 \rangle \rightarrow Lamda \ \langle Here \ x_2 = x_1 \rangle \quad (3)$$

$$\langle A \downarrow x_1 \uparrow x_2 \rangle \rightarrow \langle MULOP \rangle \ \langle Factor \downarrow x_1 \uparrow x_3 \rangle$$

$$\langle A \downarrow x_3 \uparrow x_2 \rangle \quad (4)$$

In the production (4), the inherited attribute of A on LHS is x_1 whereas that on RHS is x_2. This inherited attribute to A changes every time 'Factor' is parsed. The same effect is to be achieved by proper extension to EAGs for production (1). This is accomplished as follows:

Kleene operator ({ }*) is considered as a non-terminal attributed symbol with input and output attributes. Input attributes are always inherited and output attributes are always synthesized. After each parse of Kleene operator every input attribute with attribute position i takes the value of the output attribute with the same attribute position i.

In our example the input attribute is x_1 and output attribute is x_3. After each parse of kleene operator the assignment $x_1 := x_3$ takes place.

Production (1) can be equivalently written as:

$$\langle Term \downarrow x \uparrow x_2 \rangle \rightarrow \langle Factor \downarrow x \uparrow x_1 \rangle \langle Here\ x_2 = x_1 \rangle$$

$$\downarrow x_2 \{ \langle MULOP \rangle\ \langle Factor \downarrow x_2 \uparrow x_3 \rangle\ \}^* \uparrow x_3$$

This extension to EAG provides a concise notation for expressing Right Regular Part CFGs in attribute grammar rules thus improving readability. Pascal syntax contains many RRPCFG productions. This extensions preserves all the basic properties of EAG.

2.4.3 DEVIATIONS FROM EAG

In an EAG there may be different paths for evaluating
the attributes depending upon the value of one or more
attributes. This may give rise to a large number of prod-
uctions.

Example

(1) <Domain type ↑ symbol table ↓ local table ↓ ptr names

↓ptrnamesdef ↑ address ↓ typevar flag

↑local tableout ↓ level> →

Ident ↓ Name   <Required Name ε (Symbol table / Local table)>

<Here Address = Address(Name)>

<where Address.Mode.Kind.Kindtag = typee>

<Here localtableout = local table>

(2) <Domain type ↓ symbol table ↓local table ↓ ptr names

↑ ptrnamesdef ↑ address ↓ True

↑ localtableout ↓ level> →

Ident ↑ Name <Require Name does not belong to

(Symbol table / local table)>

<Here local tableout = local table Ø {Name → (typee;

void; level)} >

<Here ptr namesdef = ptrnames⁀name>

<Here address = address(Name)>

(3) <Demaintype ↓Symboltable ↓localtable ↓ptrnames

      ↑ptrnamesdef ↑address ↓false ↓localtableout

      ↓level> →

Ident ↑Name <Required Name does not belong to

(Symboltable/Localtable)>

              <ERROR ...>

We have reduced the number of productions to a
minimum by using explicit selection of evaluation paths
taking the values of one or more attributes as a criterion
for selection.

Thus in the modified EAG, the three productions of
the above example are merged into one as:

<Domain type ↓ Symboltable ↓ localtable ↓ ptrnames

     ↑ptrnamesdef ↑address ↓typevarflag

    ↑localtableout ↓ level> →

Ident ↑ Name <Here Flag = Name ε (Symboltable/Localtable)>

Flag ?

    TRUE : <address = address (Name)>

        <where address.Mode.Kind.Kindtag = typee>

        <localtableout = localtable>

    FALSE:

        Typevarflag ?

            TRUE: <localtableout = localtable

                   U {Name → (typee;void;level)} >

                <ptrnamesdef = ptrnames⌐Name>

                <Here address = address(Name)>

            FALSE: <ERROR ... >

CHAPTER III

IMPLEMENTATION OF STATIC CONTEXT SYNTHESIZER

In this chapter we describe actual implementation of static context synthesizer whose formal specification in L-EAG has been derived in the previous chapter. We derive transformation ruls to convert L-EAG specification into a Pascal program. We have adopted Pragmatic Transformation Strategy (PTS) in the actual implementation. Symbol Table contains the context synthesized during declaration part of Pascal programs. Symbol table construction and various domains used in the process are presented in this chapter.

3.1 CORRESPONDENCE BETWEEN ATTRIBUTE GRAMMARS AND PASCAL PROGRAMS

In this section we attempt to provide transformation rules to convert our formal specification into equivalent PASCAL program. The correspondence between a LL(1) CFG specifying context free syntax of PASCAL and an equivalent PASCAL program has been established in [DAT 81]. We extend this correspondence to produce transformation rules for converting attribute grammar specification, specifying syntactic and context sensitive issues of PASCAL, into an equivalent program.

The attribute grammar is expressed in modified extended BNF (MEBNF). The input to transformation rules is MEBNF specification of our attribute grammar. We had to modify EBNF so as to include explicit attribute structure & evaluation

rules of the attribute grammar. In an MEBNF production we use <u>attributed symbols</u> and specify explicit attribute evaluation rules. Analogus to the local extensions to EAG described in Section 2.4.2. the kleene operator ({ }) in MEBNF is associated with input (inherited) and output (synthesized) attributes with the same semantics. We follow the following conventions in MEBNF specification:

Attributes associated with symbols and Kleene operator in MEBNF have the following semantics: IA = Set of inherited attributes and SA denotes the set of synthesized attributes. Each production in MEBNF contains some "rules". The non-terminal "production" in MEBNF is associated with a set of semantic functions F_0^p, F_0^e, F_i^t ($i = 0, 1, \ldots$) and F_k^f ($k = 0, 1, \ldots, 7$). Some of these functions may be null. These semantic functions take values of eertain attribute variables as input and produce the value of an attribute variable as output. Superscript on a function signifies the name of nonterminal on the LHS of a MEBNF production in which this function is invoked. For each instance of "production" the set of semantic functions associated is refreshed to a new set derived from the corresponding production in attribute grammar.

MEBNF productions are described below:

Representation of abstract data type in Data
Structures of PASCAL.

Abstract Data Type	Definition/ Operator	PASCAL Representation
Cartesian Product	P=(s1:T1,...,sn:Tn)	P- RECORD s1:T1; . . . sn:Tn END;
	Selection	P.s1
Discriminated Union	U= (sel):SELECTOR; (s1:(L1:T1) . . . sn:(tn:Tn))	SELECTOR-(s1,...,sn); U= RECORD CASE sel:SELECTOR of s1:(t1:T1); . . sn:(tn:Tn) END;
	Selection	IF sel = s1 then U.t1 is accessible
Maps	M = { U --> R }	M = ^ M1; M1 - RECORD d : D ; r : R ; lptr , rptr : M END; StackofM = ARRAY [0..MAX] of M
	Membership	If m1 is of type M then membership is defined by the procedure SEARCHLOCTAB or SEARCHREFTAB . If m1 is of type StackofM then member- ship is defined by procedure SEARECHSYMTAB or SEARCHREFSYMTAB.
	Address	Same as membership

	Disjoint Union	Let d:D and m:M . Disjoint Union is implemented by using two Procedures,SEARCHLOCTAB and SEARCHREFTAB. If not found then ENTERLOCTAB (PUTREFTAB) else ERROR.
	Modify	Let d:D and r:M. SEARCHSYMTAB IF found THEN BEGIN delete(d--'r); enter (d -- r); end ELSE WRITE('Compiler Error')
	Overridden	Let m1:stackofM and m2:M. Let l = top of m1. Then m1/m2 is the new value of m1 after putting m2 on top of the stack,i.e.,l+1 level, and l is made equal to l+1 (m][l+1]:=m2[1:- l+1)
Sequence	S - D*	s - ^ S1; S1 - RECORD d:D; nextptr:S END;
	Append first , Append Last , Delete	List Operators
	CAR , CDR	LISP Operators

		Override	For this purpose once
			'StackofS is defined,
			'StackofS=ARRAY[0..MAX] of S
			Override operation is
			similar to that for more.
		Occurrence Test	If a∈C and a∉D then
			SEARCHLIST or SEARCHLICIAS
			defines occurrence of J in
			A2. On the other hand if
			a∈?:StackofS then the test is
			same as the membership test
			for the sets.
Set	S = { D }	0 = S1;	
		S1 - RECORD	
		d:D; nextptr:S	
		END;	

```
*********************************************************
```

Since it has been implemented as list,therefore
list operators defined in Sequence hold in this
case also.

NOTE: Identifiers in small and capital letters are assumed
to be different.

$\langle$Prodlist$\rangle \rightarrow \langle$Production$\rangle$ { , $\langle$Prodction$\rangle$ }

$\langle$Production$\rangle \rightarrow \langle$NTSYM IA, SA$\rangle$ "$\rightarrow$" $\langle$Exp IA, SA$_1\rangle$

$\qquad\qquad$ Rule (SA = F_0^p(IA,SA$_1$))

$\langle$Exp IA,SA$\rangle \rightarrow \langle$Term IA,SA$\rangle$ {" | "$\langle$Term IA,SA$\rangle$ }

$\langle$Term IA,SA$\rangle \rightarrow \langle$Factor IA,SA$_1\rangle$ Rule (IA$_2$=F_0^t(IA,SA$_1$))

$\qquad$ Rule(SA$_M$ = SA$_1$) (Rule (I = 0)

$\qquad$ +IA$_2$ {$\langle$Factor IA$_2$,SA$_2\rangle$ Rule(SA$_M$=SA$_M$ U SA$_2$)

$\qquad\qquad$ Rule(i = i+1)} + F_i^t(IA,SA$_M$)

$\langle$Factor IA,SA$\rangle \longrightarrow$ "("$\langle$Exp IA,SA$\rangle$")"

$\qquad\qquad$ |"["$\langle$Exp IA,SA$\rangle$"]"

$\qquad\qquad$ | (Rule(IA$_1$ = F_0^f(IA))

$\qquad\qquad$ + IA$_1$"{" Rule IA$_M$ = F_1^f(IA,IA$_1$)$\langle$Exp I$_{AM}$,

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ SA$_1\rangle$

$\qquad\qquad$ Rule SA$_2$ = F_2^f(SA$_1$)"}" + SA$_2$)

$\qquad$ |(Rule(SA$_1$ = F_3^f(IA)) $\langle$Terminal SA$_2\rangle$

$\qquad$ Rule (SA = F_4^f(IA,SA$_1$,SA$_2$)))

$\qquad$ |(Rule(IA$_1$ = F_5^f(IA)) $\langle$Nonterminal IA$_1$,SA$_1\rangle$

$\qquad$ Rule(SA = F_6^f(IA,SA$_1$)))

Now we describe correspondence between an attribute
grammar, expressed in MEBNF, and equivalent PASCAL program.

TRANSFORMATION RULES

MENBF Production	Condition	Pascal program segment

1. `<Prodlist>=> <Production>` | — | `P(Prodlist)=P(Production)`
   `{","<Production>}` | | `{ P(Production) }`

2. `<Production> => <NTSYM IA,SA>` | | `P(Production)='PROCEDURE'Name(NTSYM)`
   `"─→" <Exp IA,SA_1>` | | `'('PARA(IA) PARA(SA)`
   `Rule(SA=F_0^P(SA_1,IA))` | | `'ACCFSYS,FSYS:SETOFSYS);'`

   | | `'BEGIN'`
   | | `P(Exp IA,SA_1) ';'`
   | | `EVALUATE (SA=F_0^P(SA_1,IA))`
   | | `'END;'`

3. `<Exp IA,SA> => <Term IA,SA>` | No. of Terms | `P(Exp IA,SA) = P(Term IA,SA)`
   `{"|" <Term IA,SA>}` | `= 1` |

   | No. of Terms | `P(Exp IA,SA) = 'CASE SYM OF'`
   | `> 1 and` | `*LIST (DIRECTOR(Term))':'' 'BEGIN'`
   | `Exp doesnot` | `P(Term,IA,SA)`
   | `derive LAMBDA` | `'END'`

   | | `{';'LIST(DIRECTOR(Term))':'; 'BEGIN'`
   | | `P(Term IA,SA) 'END'}`

* Refer [Dat 81]

No. of Terms
>1 and
Exp derives
LAMBDA

```
P(Exp IA,SA) = 'CASE SYM OF'
               LIST(DIRECTOR(Term))';' 'BEGIN'
                         P(Term IA,SA)
                         'END'
               {';'LIST(DIRECTOR(Term))':' 'BEGIN'
                         P(Term IA,SA) 'END'}
               'OTHERS : BEGIN
               *ERRORSET(' IRST(Exp)'),
                         EVALUATE(SA=F_o^e(IA))
                         'END'
               'END'
```

---

4. $<\text{Term } IA,SA> \Rightarrow <\text{Factor } IA,SA_1>$
   $\text{Rule}(IA_2 = F_o^t(IA,SA_1))$
   $\text{Rule}(SA_M = SA_1)$
   $\text{Rule}(i = 0)$
   $\uparrow IA_2\{<\text{Factor } IA_2,SA_2>$
   $\text{Rule}(SA_M = SA_M \cup SA_2)$
   $\text{Rule}(i=i+1)\} \uparrow F_i^t(IA,SA_M)$

```
P(Term IA,SA)=
   P(Factor IA,SA)';'
   EVALUATE(IA_2 = F_o^t(IA,SA_1)';'
   EVALUATE(SA_M=SA_1)';'
   EVALUATE(i=0)
   {';' P(Factor IA_2,SA_2)';';'
   EVALUATE(SA_M=SA_M U SA_2)';'
   EVALUATE(i=i+1),
   EVALUATE(IA_2= F_i^t(IA,SAM)
```

---

5. $<\text{Factor } IA,SA> \Rightarrow$
   (a) $'('<\text{Exp } IA,SA>')'$
   (b) $'['<\text{Exp } IA,SA>']'$

```
P(Factor IA,SA) = P(Exp IA,SA)
P(Factor IA,SA) =
   'IF SYM IN ('DIRECTOR(Exp)')'THEN'
   'BEGIN'
   P(Exp IA,SA)
   'END'
   'ELSE BEGIN' EVALUATE(SA=F_7^f(IA))'E
```

* Refer [Dat 81]

(c) Rule $IA_1 = F_o^i(IA)$

IA_1 : {'Rule($IA_M = F_{li}^f(IA, IA_1)$)}

$\langle Exp\ IA_M,\ SA_1\rangle$

Rule($SA_2 = F_2^f(SA_{ll})$)')'} ' $+ SA_2$

---

0 or more
times
($*$ operation)

P(Factor IA,SA) =

EVALUATE $(IA_1 = F_o^f(IA))$

'WHILE' SYM IN ('DIRECTOR(Exp)') D<

'BEGIN'

EVALUATE($IA_M = F_1^f(IA, IA_1)$)';'

P(Exp IA_M, SA_1) ';'

EVALUATE $(IA_1 = F_2^f(SA_1))$

'END' ';'

EVALUATE $(SA = IA_1)$

---

1 or more
times
(+ operation)

P(Factor IA,SA) =

EVALUATE($IA_1 = F_o^f(IA)$)

'REPEAT'

EVALUATE $(IA_M = F_1^f(IA, IA_1))$';'

P(Exp IA_M, SA_1) ';'

EVALUATE $(IA_1 = F_2^f(SA_1))$

'UNTIL NOT (SYM IN('DIRECTOR(Exp)'))';

EVALUATE $(SA = IA_1)$

---

d) Rule $\langle SA_1 = F_3^f(IA)\rangle$

$\langle$Terminal $SA_2\rangle$

Rule $(SA = F_4^f(IA, SA_1, SA_2))$

P(Factor IA,SA) =

EVALUATE $(SA_1 = F_3^f(IA))$ ';'

'IF' SYM =' Terminal 'THEN'

'BEGIN'

EVALUATE(SA=F_4^f(IA,SA_2,SA_1))';'

'LEXANALYSE' ;

'END'

'ELSE' ERROR (ORD('Terminal'))'

29

(e) Rule $(IA_1 = F_5^f(IA))$
<Nonterminal IA_1, SA_1>
Rule$(SA = F_6^f (SA_1))$

P(Factor IA, SA) =
EVALUATE $(IA_1 = F_5^f(IA))$';'
Name(Nonterminal)'('ACTUALPARA(IA_1)
ACTUALPARA(SA_1)
'ACCFSYS,FSYS)↓;'
EVALUATE $(SA = F_6^f(IA, SA_1))$

Routines PARA and ACTUALPARA are used in transforming attributes to formal parameters and actual parameters, respectively, while generating the program segment. Parameter to these routines is a set of attribute variables. These routines are described below:

Routine	Condition	Program Segment
1. PARA(A)	A = LAMBDA (null)	
	A = IA(list of inherited attributes)	Name(CAR(A))';' DOMAINTYPE(EVALDOM(CAR(A)))';' PARA (CDR(A))
	A = SA (List of synthesized attributes)	'VAR' Name(CAR(A))':'DOMAINTYPE(EVALDOM(CAR(A)))';' PARA(CDR(A))
2. ACTUALPARA(A)	A = LAMBDA (null)	
	A is nonempty	Name (CAR(A))',' ACTUALPARA(CDR(A))

Assumptions and Terminology used in Deriving Transformation Rules:

1. Let NT be an attributed symbol in MEBNF. P(NT) signifies the program segment corresponding to the symbol NT of MEBNF.

2. The routine EVALUTE generates a program segment for evaluating the expression given in its argument. EVALUATE is used for evaluation of an attribute encountered during passing of an attribute grammar production. Appropriate semantic function, specified, in the argument to EVALUATE, is invoked during the attribute evaluation.

3. Let A be an attribute variable. DOMAINTYPE(EVALDOM(A)) is a function which returns the DOMAIN TYPE of the attribute variable A.

4. We have not provided the rules required for producing the list of declarations local to each generated procedure for storing the attributes in a production that are not associated with the left hand side nonterminal.

5. Let L be the set of accessible local attributes, as considered in (4), i.e., No attribute in L is associated with the LHS nonterminal in a production. Invocation of any semantic function assumes that L is present in the domain of this function.

3.2 PRAGMATIC TRANSFORMATION STRATEGY (PTS)

Here, we describe transformation strategies which we have followed while converting our formal specification into an equivalent PASCAL program. These strategies are highly suitable for pratical implementations and are aimed at higher run-time efficiency by reducing memory requirements. The ensuing discussion assumes that the language chosen for implementation is block structured.

The transformation rules derived, for converting an attribute grammar specification into Pascal program, in previous section suggest that an inherited attribute can be treated as a VALUE parameter and a synthesized attribute to a non-terminal as a VAR parameter to the corresponding procedure in Pascal. In practice the transformation rules derived in previous section suffer from lot of drawbacks. The drawbacks are as listed below:

(a) Most of the procedures require very large number of parameters.

(b) Static nesting structure of the resulting program is not taken into account. The transformation rules do not give any information about it.

(c) If proper block structuring is used, many identifiers may have unique use in their scope. But as these identifiers are local to all the procedures (in which they are used) they require a lot of memory space.

(d) Attribute variables for local use in a production (not passed as either inherited or synthesized attribute to any symbol) are not taken into account.

We have circumvented these drawbacks by adopting a Pragmatic Transformation Strategy (PTS), described below:

(1) We have derived static nesting structure of the resulting program based on its CALL GRAPH. We make use of static nesting structure in converting formal specification in L-EAG into program.

(2) For each attribute to a nonterminal in the left hand side of an L-EAG production the following actions are taken with due consideration to various factors:

(i) The identifier corresponding to this attribute should not be declared as a paramater to the procedure corresponding to the nonterminal if the following holds:

(a) the use of the identifier is unique in this procedure

(b) the identifier use is unique in the siblings of this procedure and in the ancestors of this procedure encountered in the scope of the identifier.

(ii) If this attribute is inherited and if there is no use of it at all the places the procedure corresponding to the nonterminal has been called **then** the identifier corresponding to this attribute can be declared as VAR parameter of

this procedure. If, in addition, there exists a synthesized attribute of the same domain as that of the above mentioned inherited attribute, then the declaration of the identifier corresponding to the synthesized attribute is considered to be merged with the declaration of the corresponding inhe ited attribute identifier.

(iii) If the identifier corresponding to this attribute has not been made global so far and if its use is unique in all the subtrees of the parent of the procedure corresponding to this non-terminal then the identifier corresponding to this attribute should be declared as local to the parent procedure.

(3) An identifier should be made global to the whole program if its use is unique throughout the program and it has been used very frequently in the whole program.

(4) All the attribute variables needed for local use in production, not passed as inherited or synthesized attribute to the nonterminal in LHS of a production, are declared as local variable in the block of the procedure corresponding to the LHS nonterminal.

3.3 PROGRAM SEGMENT GENERATED BY TRANSFORMATION RULES

In this section we illustrate Transformation Rules and PTS through an example. A production in our formal specification (L-EAG) is taken. We obtain an equivalent Pascal program segment first only by application of transformation rules than by following Pragmatic Transformation Strategy alongwith it.

As an example we take the following attribute grammar production:

p:

<TYPEDEFPART ↓ Globaltable ↓ Localtableconst ↓ Filestocheck ↓blockinfo ↓ level ↑ localtabletype>

  → LAMBDA <Here Localtabletype = Localtableconst> |

    'TYPE' <Here Tabledefin = Localtable and Ptrnames=NIL>

   ↓ Ptrnames ↓ Tabledefin {<TYPEDEF ↓ Globaltable ↓Tabledefin

                      ↓Filestocheck ↓ Blockinto ↓ level

                      ↑Tabledefout ↓ ptrnames

                      ↑ptrnamesout> ';'

                   }$^+$ ↑ Ptrnamesout ↑Tabledefout

  <CHECKFORPTRNAMES ↓Globaltable/Tabledefin ↓Ptrnames>

  <Here Localtabletype = Tabledefin>

The following program results when transformation
rules as given in section 7.1 are applied to this production
of our attribute grammar.

```
procedure TYPEDCLPART(GLOBALTABLE : SYMBOLTABLETYPE;
                      LOCALTABLECONST :
                            PTRENVIRONMENT;
                      FILESTOCHECK : PTRNAMELSTTYPE;
                      BLOCKINFO : BLOCKINFOTYPE;
                      LEVEL : INTEGER;
                      LOCALTABLETYPE :
                            SYMBOLTABLETYPE;
                      ACCFSYS,FSYS : SETOFSYS);
    var TABLEDEFIN : PTRENVIRONMENT;
        TABLEDEFFOUT : PTRENVIRONMENT;
        PTRNAMES : PTRNAMELSTTYPE;
        PTRNAMESOUT : PTRNAMELSTTYPE;
    begin
        case SYM of
            TYPESY :
                    begin
                        TABLEDEFIN := LOCALTABLECONST;
                        PTRNAMES := nil;
                        if SYM=TYPESY
                        then
                            begin
                                LEXANALYSE
                            end
                        else ERROR(ORD(TYPESY),GLOBAL);
                        repeat
                                TYPEDEF(GLOBALTABLE,
                                        TABLEDEFIN,
                                        FILESTOCHECK,
                                        BLOCKINFO,LEVEL,
                                        TABLEDEFOUT,
                                        PTRNAMES,
                                        PTRNAMESOUT,
                                        [SEMICOL],
                                        FSYS+
                                            [SEMICOL,IDENT,
                                                EQ] +
                                                TYPEDENBEGSYS);
                                PTRNAMES := PTRNAMESOUT;
                                TABLEDEFIN := TABLEDEFOUT;
```

```
                    if SYM -SEMICOI
                    then
                        begin
                            LEXANALYSF
                        end
                    else ERROR(ORD(SEMICOL),
                                GLOBAL)
                  until not(SYM in
                            (TYPEDENBEGSYS+
                                [IDENT,EQ,
                                    CASESY ]))
                  LOCALTABLETYPE := TABLEDEFIN;
                  GLOBALTABLE[LEVEL+1] . SYMENVIRON
                                  := LOCALTABLETYPE;
                  CHECKFORPTRNAMES(GLOBALTABLE,
                                    PTRNAMES);

              end;
          OTEHRS :
              begin
                  LOCALTABLETYPE :=
                      LOCALTABLECONST;
              end
        end
    end;
```

   It is evident from the program segment generated
according to exact transformation rules that a lot of
parameters and local variables need to be declared, resulting
in unusually large memory requirements and run-time
inefficiency.

   If we apply transformation rules along with the PTS
described in section 3.2,the resulting program segment is:

```
procedure TYPEDEFPART(ACCFSYS,FSYS : SETOFSYS);
        begin
            case SYM of
                TYPESY :
                    begin
                        if SYM=TYPESY
                        then
                            begin
                                LEXANALYSE
                            end
                        else ERROR(ORD(TYPESY),GLOBAL);
                        repeat
                            TYPEDEF([SEMICOL],
                                    FSYS+
                                        [SEMICOL,IDENT,
                                        EQ]+
                                        TYPEDENBEGSYS);
                            if SYM=SEMICOL
                            then
                                begin
                                    LEXANALYSE
                                end
                            else ERROR(ORD(SEMICOL),
                                        GLOBAL)
                        until not(SYM in
                                (TYPEDENBEGSYS+
                                    [IDENT,EQ,
                                    CASESY]));
                        CHECKFORPTRNAMES(PTRNAMES);
                    end;
                OTHERS :
            end;
        end;
```

All the parameters (except ACCFSYS and FSYS) and
local variables have been made global to the procedure
TYPEDEFPART . TABLEDEFIN and TABLEDEFOUT are merged to one
global variable which is SYMENVIRON component of top of the
stack of SYMBOLTABLE. Similarly PTRNAMES and PTRNAMESOUT
are merged to one global variable PTRNAMES . Since
LOCALTABLE is always put on the top of the stack of
SYMBOLTABLE , the overridden operation is automatically
implemented.

3.4 DOMAINS USED FOR SYMBOL TABLE : GENERAL STRUCTURE

SYMBOL TABLE is a domain consisting of static context synthesized during the declaration part of Pascal program. Various component domains are used, alongwith the composition rules described in Section 2.4.1, to synthesize the complex domain, namely, SYMBOL TABLE. In this section we describe the significant underlying domains used in synthesizing the context. We present these domains in top down manner, i.e., first the constituent domains of the SYMBOL TABLE followed by the component domains of these constituents and so on down the hierarchy.

3.4.1 SYMBOL TABLE

It is declared as stack of quadruples. The level of stack corresponds to local context of the Block encountered at that nesting level.

SYMBOL TABLE = Stack of <BLOCKINFO : BLKINFO; SYMENVIRON :

ENVIRON; REFTABINFO : REFTABLEINFO;

LABELINFO : LABINFO>

BLKINFO contains information about the TYPE (Proc, func, main or record) of the Block and the qualified name of the Block.

SYMENVIRON: is the context built through the identifiers declared in the Block.

REFTABINFO contains the procedure identifiers declared and used in the block alongwith the related attributes used in solving the problem of ALIASING [Sgh 81].

LABELINFO is the context built through the labels declared in the block.

3.4.2 BLKINFO

BLKINFO = <BLKNAME : BLKNAME TYPE; BLKTYPE:BLOCKTYPE>

BLKNAME is the qualified name of the block.

BLKTYPE is the TYPE of the block (i.e., Proc., Func, Main or Record)

BLKNAMETYPE and BLOCKTYPE are described in Appendix 1 and are self explanatory.

3.4.3 ENVIRON

It denotes mapping from NAME to MODE.

NAME denotes the name of an identifier declared in the block. MODE describes the attributes associated with the identifier.

3.4.4 MODE

MODE = <MODEKIND:KIND; MODETYPE:TYPE; IDLEVEL:LEVEL>

MODEKIND reflects the place of declaration of the identifier in the declaration part of a Pascal program (i.e., CONST, TYPE, VAR or PROC/FUNC declaration). MODETYPE is discussed in the next subsection.

IDLEVEL is the level of the block in which the identifier is declared.

3.4.5 KIND

The component domains of KIND are given in Appendix 1.
We briefly present some of the non-self-explanatory domains
and selectors.

CONTVAR: It is a selector whose use is specific to
FOR LOOP control variable.

EXPKIND: An actual parameter is an EXPRESSION. But
if a formal parameter is a VAR parameter then the corres-
ponding actual parameter must be ENTIRE VARIABLE. Thus
the processing of an expression which returns its MODE,
the MODEKIND is set equal to EXPKIND if the expression is
not an ENTIRE VARIABLE.

BOUNDID: It denotes the KIND of identifiers which are
declared as index variables in the indextype declaration
of CONFORMANT ARRAY SCHEMA. These identifiers are treated
as constants though their value is not known at compile
time.

Example

Type LIM = 1..3;

Procedure Test(... x:ARRAY[M .. N : LIM] OF CHAR; ...);

Here the KIND of M and N is BOUNDID. Assigining reference
to M and N is not allowed.

LOCALFUNC: Function name is treated as local variable inside the function block. The kind of this variable is made LOCALFUNC so as to allow assigning references to it.

PROCPLAN is an attribute associated with Procedure/ Function identifiers. It is an ordered list of TYPES of formal parameters.

If the kind of an identifier is FWDPROC or FWDFUNC then an additional attribute PROCENV is associated with it. PROCENV consists of local context built during processing of formal parameter list of "forwarded" function or procedure. When the block of the 'forwarded' procedure/function is encountered, the corresponding PROCENV is pushed onto top of the SYMBOL TABLE.

FIELD: If an identifier is declared as a field of a Record structure then there is an attribute KINDTAGLIST associated with it. The KINDTAGLIST contains a list of case tag variables alongwith the values they must have so that accessibility of this field of the record is valid. It helps in implementing variant part of a record structure as a discriminated union.

3.4.6 TYPE

TYPE = <TYPEACT : TYPEACTUAL; FILESELECT : BOOLEAN;
      FATHER PACKING : PACKING TYPE>

TYPEACT consists of attributes about the TYPE of an identifier.

FILESELECT indicates, if any, component of the TYPE of an identifier is of FILETYPE. An assigning reference to a variable whose TYPE has a component of FILETYPE (i.e., FILE SELECT = TRUE) is not allowed.

FATHERPACKING: Packing information of TYPE of an identifier is stored as FATHERPACKING for each of its component. This helps in implementing the following rule: "Component of a packed structure should not be passed as an actual parameter if the corresponding formal parameter is a VAR parameter.

3.4.7 TYPEACTUAL

Its component domains are described in Appendix 1. Here we discuss some of the non-self-explanatory componwnt domains and its selectors.

(a) ORDINAL: This selector indicates that the identifier declared is of ordinal type. Two attributes ORD (domaintype is ORDTYPE) and RANGE (domain is RANGETYPE) are associate with it. ORD classifies the exact ordinal type (i.e., Integer, Char, Boolean or Enumeration). RANGE defines the set of values the identifier can be assigned to.

```
ORDTYPE = (ORDTAG:ORDTAGTYPE;(INT | CHARR| BOOLEAN
            |ENUM(ENUMLIST:NAMELSTTYPE; ENUMCOUNT:INTEGER))
RANGETYPE = (RANGETAG:RANGETAGTYPE; (ALL |SUBRANGE
            (FRANGE:INTEGER; LRANGE:INTEGER) | VALUED))
```

If an identifier is of enumeration type than two attributes ENUMLIST and ENUMCOUNT are associated with it. ENUMLIST is an ordered list of all enumerated constants encountered in the declaration of the identifier and ENUMCOUNT denotes number of such enumerated constants. Each enumerated constant encountered in the enumeration list is also associated with the two attributes ENUMLIST and ENUMCOUNT. For every CONSTANT the selector chosen, in the domain RANGETYPE, is VALUED. The value of an enumerated constant is its ordinal position in the enumeration list. Two objects of enumeration type or the enumeration constats are compatible only if the ENUMLIST associated with both of them is identical.

    (b) ARRAYY: This selector is used for identifiers of ARRAY TYPE. It has three associated attributes PACKING, INDEXTYPE and ELEMENTTYPE.

Example

    TYPE ONE = ARRAY [1..5, 1..10] OF CHAR;

       TWO = ARRAY [1..5] OF

                ARRAY [1..10] OF CHAR;

Types ONE and TWO are equivalent according to ISO definition of PASCAL. In our scheme all identifiers declared in the form of type ONE are converted to the equivalent representation of type TWO.

In the above example TYPEACTUAL of type ONE will be stored, in the same manner as that of type TWO, as follows:

    ARRAY(UNPACKED, INDEX1, ELEMENT1)

where

    INDEX1 = ORDINAL (INT, SUBRANGE(1,5))

and

    TYPEACTUAL of ELEMENT1 is stored as

    ARRAY(UNPACKED, INDEX2, ELEMENT2)

where

    INDEX2 = ORDINAL (INT, SUBRANGE(1,10))

and

    TYPEACTUAL of ELEMENT2 is stored as:

    ORDINAL ((CHARR, ALL)

Exactly the same scheme is used for CONFORMANT ARRAY SCHEMA.

(c) <u>RECORDD</u>: This selector is used for identifiers whose TYPE is Record type. Domain associated with this selector is a cartesian product of four component domains PACKINGTYPE, LAYOUTTYPE, ENVIRONMENT and BTMPTRTYPE PACKINGTYPE gives information about packing of the record structure. Attributes associated with the rest of the domains are described below:

(i) <u>LAYOUT</u>: It consists of list of TYPEACTUALs of the field of the record encountered in the fixed part of the record. Two record types are compatible if their corresponding LAYOUTs are compatible.

(ii) <u>ENVIRON</u>: It consists of local context synthesized by the fields declared in the record structure. Any identifier in this environment can be accessed only after proper qualification by the record name.

(iii) <u>BTMPTR</u>: This is used only in case of type compatibility checking for the actual parameters of the system defined procedures NEW and DISPOSE.

BTMPTR is a nested list structure, the nesting of a list corresponds to the nesting of the variant part in the Record. At each nesting level, the list consists of casetagtype and the case constants encountered in the variant part nested at this nesting level in the record structure. Each case constant in this list contains a BTMPTR which is prepared in the variant part encountered in the variant corresponding to this case constant.

```
TYPE LIM = 1 ... 3; ENUM = (SUN, MON)
       REC = RECORD
                CASE L : LIM OF
                   1 : ( ) ;
                   2 : (CASE Bool:BOOLEAN OF
                           TRUE:(CASE ENUM OF
                                   SUN:( ) ;
                                   MON:(RAM:INTEGER));
                           FALSE:(CASE B: BOOLEAN OF
                                   TRUE : ( );
                                   FALSE : ( )));
                   3 : (CASE BOOLEAN OF
                           TRUE :( );
                           FALSE:( ))
                END;
```

BTMPTR synthesized for this record is presented below in

the form of nested list structure.

TYPEACTUAL OF LIM	1	2	3
	NIL		

TYPEACTUAL OF BOOL	Ordinal value of TRUE	Ordinal value of FALSE

TYPEACTUAL of BOOLEAN	Ordinal value of TRUE	Ordinal value of FALSE
	NIL	NIL

TYPEACTUAL OF ENUM	Ordinal value of SUN	Ordinal value of MON
	NIL	NIL

TYPEACTUAL of B	Ordinal value of TRUE	Ordinal value of FALSE
	NIL	NIL
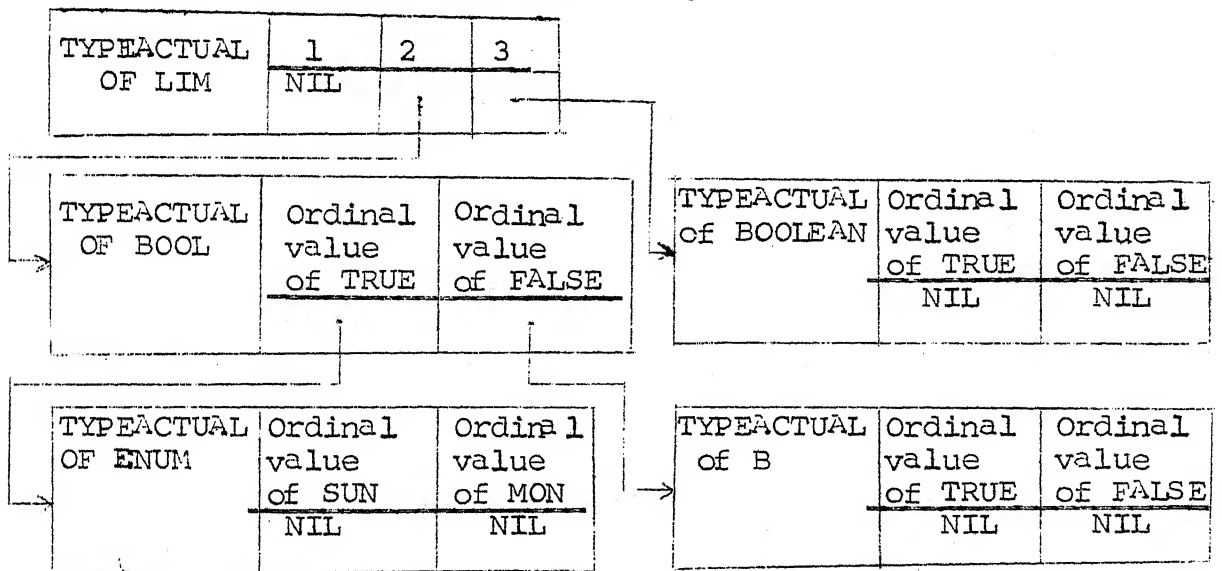
(d) UNION

Casetag of the variant part of a record is treated as a field in the fixed part of the record structure. But it is associated with an additional attribute conveying whether the variant part is a free union or a discriminate union. The information about free or discriminated union is required only while type compatibility checking of two Record types. Thus UNION is used as a selector for the casetag type while preparing LAYOUT of the record. Associated attributes with this selector are DISCRIMINATION and TAGTYPE. TAGTYPE gives TYPEACTUAL of the casetag and DISCRIMINATION indicates whether the corresponding variant part is a free union or discriminated union. While putting casetag in the local environment of the record, it is treated as any other field of the record — no information about discriminated or free union is required — hence, UNION is not used for this application.

(e) POINTERR

Since forward referencing is allowed in POINTER declaration, domaintype of a pointertype may not be known at the point of declaration. Therefore, TYPEACTUAL information for pointertypes just stores the address of identifier in the SYMBOL TABLE (i.e., address (Name $\rightarrow$ Mode)) whose TYPE is domain type of the pointer type.

(f) <u>FORMALPARCONST</u>

This selector indicates that the identifier is a formal parameter. The attribute FORMALPARVAR associated with this selector contains information required for type compatibility checking with the corresponding actual parameters.

3.4.8 <u>FORMALPARTYPE</u>

FORMALPARTYPE = (PARTAG : PARTAGTYPE;

(SIMPLEPAR(SIMPLE : SIMPLETYPE)|

DEFAULTPAR(COMSIMPLE:COMSIMPLSTYPE)|

POLYMORPH(COMSIMPLE:COMSIMPLSTYPE)|

POLYADIC(COMSIMPLE:COMSIMPLSTYPE)))

COMSIMPLSTTYPE = COMSIMPLETYPE*

The selector DEFAULTPAR, POLYMORPH and POLYADIC are used to handle the type compatibility checking of operands which may be default, polymorphic or polyadiac in case of system defined procedures, functions and operators. For user defined procedure/function parameter declaration only SIMPLEPAR is selected.

3.4.9 <u>COMSIMPLETYPE</u>

COMSIMPLETYPE = (COMSIMPTAG:COMSIMPTAGTYPE;

(VERYSIMPLE(SIMPLE:SIMPLETYPE)

FORMATTED(FORMATSIMPLE:SIMPLSTTYPE)))

The selector FORMATTED is used if formatting of the corresponding actual parameter is allowed to be specified alongwith the actual parameter. In the system defined

procedures WRITE and WRITEIN we can specify format of an actual parameter, e.g.,

WRITE (INT : 5; REEL : 10:I);

Attribute associated with the selector FORMATTED is FORMATSIMPLE from the domain SIMPLSTTYPE. FORMATSIMPLE is an ordered sequence of SIMPLE TYPEs, the header of the sequence is to be matched with the corresponding actual parameter. Rest of the sequence contains information about formatting allowed with the corresponding actual parameter. It is not an error if the format of an actual parameter is not specified and a default formatting is assumed in the absence of explicit specification.

3.4.10 SIMPLETYPE

SIMPLETYPE = <LASTPROPAGATED: BOOLEAN; CRITERIA:CRITERION;
 NXTRROP :NEXTPROPAGATION ;SIMPMODE:MODE;
 LATTICEVALUE:INTEGER>

LASTPROPAGATED: Type compatibility checking of an actual parameter may have to be performed with a TYPE derived and propagated by some previous actual parameter. Such a situation can arise only with the type compatibility checking of a system defined procedure, funtion or an operator.

Example

A := A+B - Let A be of set type then the operator PLUS requires that type of the second operand B must be compatible with the type of the first operand A.

PROPAGATION TYPE stores the type propagated from a previous actual parameter. Thus LASTPROPAGED indicates if the type compatibility checking of the corresponding actual parameter is to be performed with the PROPAGATION TYPE or the formal parameter type present in SIMPMODE.

CRITERIA: It classifies the class of type compatibility to be invoked for the matching of actual and formal parameters.

LATTICE VALUE

Type compatibility checking of formal and corresponding actual parameter returns this lattice value. It is used in Operator Identification and determining the resultant type of an operator or a function [Sgh 81].

3.4.11 REFTABLEINFO

REFTABLEINFO = [PROCINFO -> MODEOFPROC]

PROCINFO = <NAMEOFPROC:NAMEIDENT; LEVELOFPROC:INTEGER>

PROCMODTYPE = <REFLST:LSTOFLST;GLOBLST:LSTOFLST;

PROCBLKDECTAG:BOOLEAN>

REFTABLE INFO is used for solving the problem of ALIASING [Sgh 81] associated with procedures.

PROCBLKDECTAG indicates whether the block of the procedure has been passed. If this attribute value is true then GLOBLST contains all the variables which are global to the procedure and assigning references have been made to them in the statement part of the procedure. IF PROCBLKDECTAG is FALSE then REFLST contains the list of actual VAR parameters, declared global to the procedure, which have been passed during the invocations of this procedure.

3.5 EXPERIENCE IN PROGRAM CONSTRUCTION BY TRANSFORMATION RULES

Transformation rules described in Section 3.1 and Pragmatic Transformation Strategy of Section 3.2 greatly facilitated our task of program construction from the formal specification. They also helped in the correct implementation of context sensitive issues of PASCAL programs. The task of program construction was close to a mechanical process involving invocations of series of transformation rules at various stages. But still we feel that these transformation rules are not well formalized and t they do not encompass all the details pertaining to implementation. Further these transformation rules are more suitable if the formal specifications is Attribute Grammar, whereas we have specified our problem in L-EAG.

We have followed PTS very closely. With this strategy
the task of converting a L-EAG production into a program
segment is no longer localized to this production only.
PTS requires scanning of various parts of L-EAG specifi-
cation dictated by the static nesting structure of the
resulting program to accomplish this task. For example
the decision about making an attribute variable GLOBAL
requires that we establish its unique use throughout the
L-EAG and, hence, correspondingly throughout the life
time of the resulting program. It may seem to be a
lengthy and very involved task but we could follow PTS
quite mechanically and with ease.

CHAPTER IV

EPILOGUE

The aim of this project was to obtain a formal basis for the specification of the static context in PASCAL programs and provide a basis for the conversion of the specification to program. We have been motivated to use L-EAGs because of their adaptability to LL(1) parsing : LL(1) provides a general basis for the structure of the program, in which is embedded the checks of static context obtained through L-EAG specification.

Our experience is that the articulation of the L-EAG specification helped clarify several deep issues in the programming language PASCAL. The L-EAG used proved to be extremely concise at the same time completely defining the context-free and context-sensitive syntax of Pascal. The underlying context free syntax was clearly visible. The conciseness stems mainly from the freedom to choose any suitable domains and operations for the attributes, and the ability to use attribute expressions freely in the semantic rules.

With regards to rules of transformation which convert L-EAG specification to Pascal programs, the following observations are in order:

(a) Proper use of nesting of generated procedures can help in increasing the efficiency of the resultant program.

(b) Aids to perform L-EAG to L-EAG transformation so that the resultant specification is efficient with regards to its utilization of attributes would be of great value.

(c) The transformation rules that we have given pertain to Attribute Grammar and they need careful adaptation to L-EAG.

The articulation of L-EAG specification, transformation rules and Pragmatic Transformation Strategy used greatly facilitated in handcoding the program with confidence. The following observations are made in coding of this program:

(1) Most of the bugs in the resulting program originated from the improper initialization of POINTERS which were elements of newly generated objects of various domains. In retrospect, it would prove fruitful to write an allocation procedure for every dynamically creatable domain element so that the component pointers can be appropriately initialized.

(2) Use of identifier names too close to the standard names in PASCAL should be avoided as far as possible. For example we have used ORD as a field name in a record structure. Thus inside a with statement, wherein the local environment of this record can be accessed without qualification, the standard function ORD cannot be used.

(3) We have used files as a way out of the garbage collection problem whenever the end of a scope is encountered. This is because of forward references to objects yet not defined within the scope. This implementation strategy can be avoided if all newly allocated domain objects are linked with respect to their logical use. Garbage collection will merely involve the recycling of lists no longer needed. Proper implementation of this strategy would require central allocation and deallocation routines for domain elements.

(4) The ISO standard for PASCAL tightens several context-sensitive constraints imposed on a program. Almost all Pascal programs of any consequence do not satisfy the constraints imposed by ISO standard. The commonly detected errors by our processors are:

(a) Improper uses of labels and goto's

(b) Modification of WITH RECORD variables
    encountered in WITHLIST

(c) Aliasing

(d) Use of globally declared variables as
    control variables in FOR STMT.

We feel that some of these errors are reported because of the strictness of the constraints and not because of poor programming practice.

Finally though we have derived L-EAG for PASCAL and our observations regarding transformation rules etc., are specific to PASCAL, we feel that very similar strategies and much of what we have discussed is applicable to contemporary 'typed' programming languages.

REFERENCES

1. Add 80   Addyman, A.N. : A Draft Proposal for PASCAL
            ACM SIGPLAN Notices 15,4 (April 80)

2. Boc 76   Bochmann, G.V. : Semantics Evaluation from
            Left to Right, CACM, Feb. 1976.

3. Dat 81   Datta, S. : Generation of LL(1) Recursive
            Descent Parsers with ERROR Recovery from
            EBNF Specifications, M.Tech. Thesis, IIT
            Kanpur 1981.

4. DDH 72   Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. :
            Structured Programming, 1972.

5. Knu 68   Knuth, D.E. : Semantics of Context Free
            Languages, Mathematical Systems Theory 2,
            1968.

6. KR 79   Kennedy, K., Ramanathan J. : A Deterministic
            Attribute Grammar Evaluator Based on Dynamic
            Sequencing, ACM, TOPLAS 1,1 (July 1979)

7. MLB 76   Marcotty, M., Ledgard, H.F., Bochmann, G.V. :
            A Sampler of Formal Definitions, Computing
            Surveys 8, 1976.

8. Sgh 81   Singh, K. : Static Contextual Analysis of
            PASCAL Programs with L-Attribute Grammars,
            M.Tech. Thesis, IIT Kanpur, 1981.

9. Wat 79   Watt, D.A. : An Extended Attribute Grammar
            for PASCAL, ACM SIGPLAN Notices 14,2 (Feb.
            1979).

10. WM 77 Watt, D.A., Madson, O.L. : Extended Attribute
            Grammar, Report No. 10, July 1977.

APPENDIX -I

DOMAIN DEFINITION

Domains used in L-EAG specification of standard
PASCAL are described below.


```
SYMBOLTABLE  =      stack of
                    (blockinfo:BLKINFO;symenviron:ENVIRON;
                     reftabinfo:REFTABLEINFO;
                     labelinfo:LABINFO)

BLKINFO  =          (blkname:BLKNAMETYPE;blktype:BLOCKTYPE)

BLKNAMETYPE  =      (blktype:BLOCKTYPE;
                    (maintype(nameofblk:NAME)|
                     proctype(nameofblk:NAME)|
                     functype(nameofblk:NAME)|
                     recordtype(recnamelst:QUALIFYLST)))

BLOCKTYPE  =        (maintype,proctype,functype,recordtype)
                                     *
QUALIFYLST  =       QUALLSTELEM

QUALLSTELEM  =      (qfnamevaltas:NAMEVALTYPE;
                    (valueconst:(qualvalue:INTEGER)|
                     nameconst:(qualname:NAME)))

NAMEVALTYPE  =      (valueconst,nameconst)

ENVIRON  =          { NAME-->MODE }

MODE  =             (modekind:KIND;modetype:TYPE;
                     idlevel:INTEGER)
```

```
KIND =              (kindtag:KINDTAGTYPE;
                    (void!
                     const(kindconstval:CONSTVALTYPE)!
                     varr!
                     type!
                     func(plan:PLAN)!
                     localfunc(plan:PLAN)!
                     formalfunc(plan:PLAN)!
                     proc(procplan:PLAN)!
                     formalproc(procplan:PLAN)!
                     forwardfunc(funcplan:PLAN;funcenv:ENVIRON)!
                     forwardproc(procplan:PLAN;procenv:ENVIRON)!
                     field:(kindtaglst:TAGLSTTYPE)!
                     formalvar!
                     formalval!
                     boundid!
                     contvar!
                     expkind))

CONSTVALTYPE =      (constvaltag:VALUETAGTYPE;
                    (valuepresent:(value:INTEGER)!
                     valueabsent))
                              *
PLAN =              TYPEACTUAL
                                                        *
TAGLSTTYPE =        (tagname:NAME;tagcasevalue:VALLSTTYPE)
                         *
VALLSTTYPE =        INTEGER

TYPE =              (typeact:TYPEACTUAL;fileselect:BOOLEAN;
                     fatherpacking:PACKINGTYPE)

PACKINGTYPE =       (unpacked,packed)
```

```
TYPEACTUAL =        (typeactualtag:ACTUALTAGTYPE;
                    (void!
                     real!
                     ordinal(ord:ORDTYPE;range:RANGETYPE)!
                     arrayy(packing:PACKINGTYPE;
                            indextype:TYPEACTUAL;
                            elementtype:TYPE)!
                     confarray(packing:PACKINGTYPE;
                            confindextype:TYPEACTUAL;
                            elementtype:TYPE)!
                     recordd(packing:PACKINGTYPE;
                            layout:PLAN;recenviron:
                            ENVIRON;recbtmptr:BTMPTRTYPE)!
                     union(discrimination:DISCTYPE;
                            tagtype:TYPEACTUAL)!
                     sett(basetype:TYPE)!
                     nullset!
                     filee(comptype:TYPE)!
                     textfilee!
                     pointer(addr:ENVIRON)!
                     nilptr!
                     formalparconst(formalparvar:FORMALPARTYPE)))

ORDTYPE    =        (ordtag:ORDTAGTYPE;(int!char!boolean!
                     enum(enumlist:NAMELSTTYPE;enumcount:
                     INTEGER)))
                              *
NAMELSTTYPE =       NAME

RANGETYPE  =        (rangetag:RANGETAGTYPE;(all!
                     subrange(frange:INTEGER;lrange:INTEGER)!
                     valued))

BTMPTRTYPE  =       (tagtypeactual:TYPEACTUAL;varaccesslst:
                     VALUEPTRLST)
                                                    *
VALUEPTRLST =       (value:INETGER;btmptr:BTMPTRTYPE)

DISCTYPE  =         (discriminated,free)

FORMALPARTYPE  =    (partag:PARTAGTYPE;
                     (simplepar(simple:SIMPLETYPE)!
                      defaultpar(comsimple:COMSIMPLSTYPE)!
                      polymorph(comsimple:COMSIMPLSTYPE)!
                      polyadic(comsimple:COMSIMPLSTYPE)))
```

```
                              *
COMSIMPLSTTYPE = COMPSIMPLETYPE
COMSIMPLETYPE  = (comsimptag:COMSIMPTAGTYPE;
                  (verysimple(simple:SIMPLETYPE)!
                   formatted(formatsimple:SIMPLSTTYPE)))
                              *
SIMPLSTTYPE  =   SIMPLETYPE

SIMPLETYPE   =   (lastpropagated:BOOLEAN;criteria:CRITERION;
                  nextprop:NEXTPROPAGATION;simplemode:MODE;
                  latticevalue:INTEGER)

CRITERION    =   (typeequivalence,assignment,membership,
                  class,rangerelax,arrayclass,strgclass,
                  arrrangerelax)

REFTABLEINFO =  { PROCINFO --> PROCMODETYPE }

PROCINFO = (nameofproc:NAME;levelofproc:INTEGER)


PROCMODETYPE =  (reflst:LSTOFLST;globlst:LSTOFLST;
                 procblkdectas:BOOLEAN)
                         *
LSTOFLST  =     QUALIFYLST
                         *
LABINFO   =     INTEGER
                         *
MODELSTTYPE =   MODE

STACKLBLTYPE =  { INTEGER-->LBLLSTTYPE }
                         *
LBLLSTTYPE  =   INTEGER
                         *
ADRLST  =       ENVIRON

SAFETYPE  =     { addr1:ENVIRON;addr2:ENVIRON }
                                             *
LABJUMPTYPE =   (labelvalue:INTEGER;levelvalue:INTEGER)
```

APPENDIX 2

The following is a list of attribute variables used
in the L-EAG productions together with their domains:

SYMBOL TABLE : Globaltable, Symboltable, Symboltable1,
Symboltableout, Symboltablein, Symboltablemid.

BLKINFO: Blockinfo

QUALIFYLST : Globallist element, Reflistelement

QUALSTELEM : Loclstelem, curlstelem

ENVIRON   : Localtable, localtableconst, localtableconst1,
Localtableconst2, Tabledefin, Tabledefout,
Tabletype, Tabletypeindex, Recordtablein,
Recordtableout, Recordtablesection, Tabletype1,
Recordtablevariant, Tabletypeout, Proctablein,
Proctable, Proctableout, Proctable1,
Proctablemid, Proctable2, Localtablehead,
Address, Localtablevar, Localtableproc,
Functable.

MODE   : Mode, Expmode, Paramode, Simpmode

KIND   : Kind, Modekind, Calledprockind, Parentkind,
Parameterkind

KINDTAGTYPE : Kindtag, Kindl.

```
PLAN          : Layout, Layin, Layoutsection, Plan, Planin,
                Planout, Formalactualtypelist, Remformalactualtype-
                list, Formaltypeactuallistin, Formaltypeactual-
                listout

TAGLSTTYPE : Taglist, Taglistout, Taglistl

TAGLSTELEM : Taglstelement

VALLSTTYPE : Taglistvalue, Taglistvaluel, Caseconstlistl,
             Caseconstantlist, caseconstlist2, Tagvaluelist,
             Taglistvaluein, Taglistvalueout

TYPE          : Type, componentype, comptype, filedomain,
                domaintype, elementtype, recordtype

PACKINGTYPE: Parentpacking, packing, selfpacking

TYPEACTUAL : Consttype, Actualtypearray, Indextype, Actualtype,
             Tagtypeactual, Typeactual, Resulttype,
             Propagatedtypeactual, Propagationtypeactual,
             Propagationtypeactualmid, Typeactuall,
             Typeactual2

ORDTYPE       : Ordinal, Ordinall, Ordinal2

NAMELSTTYPE: Filestocheck, Remfiles, Identifierlist, Name-
             list, Remfilestocheck, Parameterlist

RANGETYPE   : Rangel, Range2

BTMPTRTYPE : Btmptr

VALUEPTRLST: Variantaccesslist, variantaccesslistin

DISCTYPE    : Discrimination
```

```
FORMALPARTYPE  : Formalparvar

COMSIMPLSTTYPE: Combinedsimplelist

COMSIMPLETYPE : Combinedsimple, combinedsimple1, combined-
                simple2

SIMPLSTTYPE    : Simplelist

SIMPLETYPE     : Simple, Simple1, Simple2

CRITERION      : Criterion

REFTABLEINFO   : Reftableblock, Reftableout, Reftablein,
                 Reftableout1, Reftableout2

REFTABLEELEM   : Reftableelementin, Reftableelementout

LSTOFLST       : Reflst, Globlst, Globvarlst1, Globvarlst2,
                 Globvarlstmid, Withlist, Globalvarlist,
                 Globallist, Withlist1, Withlist2, Withlistmid
                 Parameterlist, Parameterlistmid, Varlistin,
                 Varlistmid

MODELSTTYPE    : Actualmodes

STACKLBLTYPE   : Globallabels, Stackoflabels, stackoflabels1,
                 Stackoflabelsin

LBLLSTTYPE     : Locallabels, Decllables, Decllabels1,
                 Decllables2, Decllabelsin, Decllablesout

ADRLIST        : Ptrnames, Ptrnamesdef, Ptrnamesout,
                 Ptrnamessection
```

```
SAFETYPE        : Safe1, Safe2, Safemid, Safe3

LABJUMPTYPE     : Labelstojump, Labelstojump1, Labelstojump2,
                  Labelstojumpin, Labelstojumpout

NAME            : Procname, Funcname, Name, Idname

BOOLEAN         : Typevarflag, Fileselect, Fixpart, Formaltag,
                  Forwardflag, Presenceflag, Withtag,
                  Varaccesstag, Errinexp, Mismatch, Exitflag,
                  Errsuppress

INTEGER         : Level, Blocklevel, Withblklevel, Nestinglevel,
                  Procblocklevel, Calledproclevel, Latticevalue
```

the following is the list of A-?Ab productions for the programming language
PASCAL. These productions are pertaining to declaration part of the PASCAL programs.

The following conventions have been adopted in the following grammar text:

```
=a=   stands for inherited attribute
=s=   stands for synthesised attribute
===   stands for append (concatenation)
===   stands for disjoint append
=?=   stands for disjoint union
=v=   stands for logical OR
=^=   stands for logical AND
=u=   stands for set union
```

(1) < Program > ==> < ProgramHeading $filestocheck > ";" < PrepareEnvironment %globaltable >
                       <ChangeEnvironmentForInputOutput $filestocheck %remfiles %localtable >
                       < Program%block %localtable $localtable %remfiles >

(1.a) <ChangeEnvironmentForInputOutput $filestocheck %remfiles %localtable > ==>
        < [Input/Output]belongsToFilesToFilesToCheck $filestocheck $input %ABSENT >
        < ChangeEnvironmentForOutput $filestocheck %remfiles $(} %localtable >

(1.a.2) < ChangeEnvironmentForInputOutput $filestocheck %remfiles %localtable > ==>
          <[Input/Output]belongsToFiles $filestocheck $INPUT %PRESENT >
          <ChangeEnvironmentForOutput $filestocheck $INPUT %remfiles >
          $INPUT ==> (var;(texcfile;TRUE;FALSE);1) %localtable >

(1.a.1.b) <ChangeEnvironmentForOutput $filestocheck $inlocaltable $inlocaltable >
            <[Input/Output]belongsToFilestocheck $filestocheck $OUTPUT $ABSENT > =

```
(1.a.1.a.) < ...environment...or output $filestocheck #inlocaltable
              -inlocalin in ...==> (var;(textfile;TRUE/FALSE)];} >==>
              < ...UL;info in lajonistofilestocheck $filestocheck SOUTPUT $PRESENT >

(1.a.1.b.) <Iti...t...list...Out.oFfilestocheck $filestocheck $NAME
              ...; belongsin filestocheck > ==> [NULL

(2.a.) <Program..Param..s;1; > ==> "PROGRAM "Ident" $NAME

(2.b.) <Program...param $filestocheck > ==> "PROGRAM "Ident" $NAME [ "("
              < Programparm $filestocheck >;}; !

(3) <Programpara #filestocheck > ==> <identlist $() $filestocheck >

(4) < identlist $localtable <identifierlist > ==> "ident" $NAME1
              <here iar.f1 intersect, $w.(localtable} = NULL >
              <here identifierlist1 = [NAME1]] > $identifierlist1{! "ident" $NAME2
              <here NAME2 intersect $w.localtable} = NULL >; $identifierlisti =
              identifierlist NAME2 < here identifierlist:identifierlist1> >

(5) < Programblock $globaltable $localtable $filestocheck > ==> < Block $globaltable'
              $localtable $() $labelistojump $filestocheck $() $reftablerinal_$]> $globaltable
              < here labelstojump = NIL >

(6) <Block $globaltable $localtable $labelistojump $filestocheck $blockinfo $reftableblock
              $reftapleout > ==> (bvd) <labeldecpart $localtables>
              < Const,defvart $globaltable $localtableconst >
              <localtableconst >
              <typeDefPart $globaltable $localtable $localtableconst $filestocheck $blockinfo
              $level $localtabletype >
              < varDecPart $globaltable $localtablevar $filestocheck $blockinfo $level
              <localtablevar $reufillestocheck > < here reufilestocheck = NIL >
              <ProcFuncDecPart $globaltable $localtable $globallabels/locallabels
              labelstojump $localtableproc $level $reftableblock $reftablein
              $blockinfo >
              < Statementpart $globaltable/localtableproc(blockinfo) $globallabels $locallabels
```

```
$labelstojumproc $labelstojumpout $plockinfo $level $reftablein
$reftableconl > < where labelstojump intersect locallabels = NULL >


(7.a)  < Labelvecpart %1 > ==> NULL


(7.b)  < Labelvecpart %locallabels > ==> "LABEL"  < Label1 $labelvalue>
                                        < Here labelist1 = llabelvalue1 >
                                        , %labellist1 = labellist1
                                        / %labellist1 [ labelvalue2]  < Label1 %labelvalue2 >
                                                                      < Here locallabels=labellist1>


(8.)   < Label1 %INTVALUE > ==> "Intconst" %#INTVALUE < where 0<= INTVALUE <= 9999 >
                               CONSTDEFPART


(9.a)  < ConstDefPart $globaltable $localtable $fielstocheck $blockinfo $level
         %localtable > ==> NULL


(9.b)  < ConstDefpart $globaltable $localtable $fielstocheck $blockinfo $level
         %localtableleconst > ==> "CONST" < Here localtableleconst1:=localtable >
         $localtableleconst1 < Constdef $globaltable $localtable $fielstocheck
         $localtableleconst2 >"#"; , >+ %localtableleconst1 >
         localtable $localconst1 < Here localtableleconst1:=localtableleconst1 >


(10)   <ConstDef $globaltable $localtableleconst $fielstocheck $blockinfo $level
         %localtableleconst < (name ) (const;(Consttype;FALSE;UNPACKED);level)%>
         ==> "Ident" %NAME  < here NAME does not belong to fielestocheck >
         "=" < Constant $globaltable/localtableleconst(blockinfo) %consttype >


(11.a) < Constant $symboltable %(ORDINAL(INT;VALUFO(INTVALUE) ) > ==> "Intconst" %INTVALUE


(11.b) <Constant $symboltable %(ORDINAL(INT;VALUED(OP#INTVALUE)) ==> "Sign" %OP
         "intconst" %INTVALUE


(11.c) < Constant $symboltable %real > ==> "realconst"
```

```
(11.d)  < Constant $symboltable $REAL > ==> "Sign" $OP "Realconst"

(11.e)  < Constant $symboltable $consttype > ==> <.ConstId $symboltable $consttype >

(11.f)  < Constant $symboltable $((INT:VALUED(INT:VALUED(OP*value)) ==> "Sign" $OP
        <ConstId $symboltable $ORDINAL(INT:VALUED(value)) >

(11.g)  < Constant $symboltable $REAL > ==> "Sign" $OP <ConstId $symboltable $REAL >

(11.h)  <Constant $symboltable $ARRAY(PACK;$ORDINAL(INT;SUBRANGE(1;STRGLGTH));
        (ORDINAL(CHAR;AL))):FALSE;1(UE)) ==> "Strqconst" $STRGLGTH

(11.i)  < Constant $symboltable $ORDINAL(CHAR;VALUED(ORDVALUE)) > ==> "Charconst" $ORDVALUE

(12)    < ConstId $symboltable $constmode.type.typeactual > ==> "Ident" $NAME
        < Here construode = mode(NAME;symboltable)) >
        < Where construode.kind.kindtag=constt>

(13.a)  < Typedefpart $globaltable $localtableconst $filestocheck $blockinfo $level
        $localtableconst > ==> NULL

(13.b)  < Typedefpart $globaltable $localtableconst $filestocheck $blockinfo $level
        $localtabletype > ==> "TYPE" < Here tabledefin :=localtableconst
        and ptrnames:=[] > $ptrnames stepledefin(< Typedef $globaltable
        $tabledefin $filestocheck $blockinfo $level $tabledefout $ptrnames
        $ptrnamesout >) $tabledefin:=tabledefin < Here localtabletype:=tabledefout $ptrnamesout
        < Here localtabletype:=tabledefin >
        <Checkforptrnames $globaltable/localtable $ptrnames >

(14)    < Typedef $globaltable $localtable $filestocheck $blockinfo $level $tabletype1
        $ptrnames $ptrnamesdef > ==> "Ident" $NAME <where NAME does not belong
        to filestocheck $localtable $globaltable $localtable $filestocheck
        $blockinfo $level $tabletype $UNPACKED $type $TRUE >
```

? name belongs to ptrnames

NODE:<here address:=address:=cross*NAME[globaltable/localtable(blockinfo)]>
    <here tablelevel:=localtable and type:=address.mode.type and
    ptrnamesdef:=ptrnamesdef=[NAME] >
FALSE:< here tabletype:=tabletype @ {NAME -->(TYPE;type;level)} >

(15) < Typedenoter %globaltable %localtable %tilestocheck %blockinfo %level %tabletype
      %ptrnames %ptrnamesdef %parentpacking %type %typevarflag > ==>
      <SimpleType %globaltable %localtable %tilestocheck %blockinfo %level
      %tabletype %type %ptrnames %typevarflag > | <StructuredType %globaltable
      %localtable %tilestocheck %blockinfo %level %tabletype %ptrnames
      %ptrnamesdef %parentpacking %type %typevarflag > |
      <pointertype %globaltable %localtable %localtable %parentpacking
      %type %ptrnames %ptrnamesdef %typevarflag %tabletype %level >

(16.a) <SimpleType %globaltable %localtable %tilestocheck %blockinfo %level %localtable
      %parentpacking %type %ptrnames %typevarflag > ==>
      <idbegintype %globaltable %localtable(blockinfo) %parentpacking %type
      %ptrnames %typevarflag > | <Subrangetype %globaltable %localtable(blockinfo)
      %parentpacking %type >

(16.b) < SimpleIdtype %globaltable %localtable %tilestocheck %blockinfo %level %tabletype
      %parentpacking %type %ptrnames %typevarflag > ==> <Enumerationtype
      %globaltable %localtable %tilestocheck %blockinfo %level %tabletype >

(17.a) <idbegintype %symboltable %parentpacking %(typemode.type.typeactual;
      typemode.type.fileselect;parentpacking) > ==> "ident" %NAME
      %typevarflag
      TRUE:<here NAME does not belong to ptrnames >
      FALSE:-NULL
      < here typemode:=mode(ENV[NAME]) > <where typemode.kind.kindtag=TYPE >

(17.b) <idbegintype %symboltable %parentpacking %(ORDINAL(typeactual1.ordselect;SUBRANGE
      (mode.type;typeactual.rangeselect;typeactual1.rangeselect);FALSE;
      parentpacking) %ptrnames %typevarflag > ==> "ident" %NAME < here
      mode:=ENV[NAME] > <where mode.kind.kindtag=constt and mode.type.typeactualtag
      =UPPER1 > "." < Constant %symboltable %typeactual1 >

<where typeactual1.typeactualtag=ORDINAL> <OrdinalEquivalence
&mode.type.typeactual1.ordselect stypeactual1.ordselect %mismatch %FALSE >
<where mode.type.typeactual1.rangeselect <=typeactual1.rangeselect >

<EnumerationType $Globaltable $Localtable $FilestoCheck $Blockinfo $Level
%tabletype $parentpacking ==> #(<identlist $localtable %namelist>)<wherenamelist
parentpacking)> ==> #(<identlist $localtable $namelist)#All)#false;
intersect FilestoCheck =null>
          <EnumerationConstant $localtable $blockinfo $level $Tabletype $Namelist
                    $(enum(namelist)) $count>

.a.1 <EnumerationConstant $localtable $blockinfo $level $Localtable $[1] $Ordinal
%n ==> %nul

.a.2 <EnumerationConstant $Localtable $blockinfo $level $Localtable $u {name
--> (<const/(ordinal(ordinalvalued(n));false;unpacked);
level)) $name namelist $ordinal $n ==> name belongsto FilestoCheck
true;PreparelistincrementCount false;DonotIncrementCount
          <EnumerationConstant $localtable $blockinfo $level $Localtable1 $Namelist
$ordinal $n+1>

<SubrangeType $symboltable $parentpacking %(ordinal(actualtype1.ord;subrange(actualtype1.range,
          actualtype2.range));false;void;parentpacking)>
          ==> <Constant $symboltable $actualtype1 > ..
               <Constant $symboltable $actualtype2 > ..

<where Actualtype1.Actualtypetag = Ordinal><Constant $symboltable
%Actualtype2><where Actualtype1.Actualtypetag = Ordinal><RelaxedOrdinal
Equivalence $Actualtype1.Ordselect $actualtype2.Ordselect> $mismatch>
<where Actualtype1.Rangeselect <= Actualtype2.Rangeselect>

<StructuredType $Globaltable $Localtable $FilestoCheck $blockinfo
$level %tabletype $ptr.names $Parentpacking %TYPE=
==> <FieldPacking:=unpacked><packed $Packing<Unpackedstructtype
$globaltable $localtable $FilestoCheck $blockinfo $level %tabletype
$ptr.names $ptr.names $parentpacking $packing %TYPE> %typevarflag

<UnpackedStructType $Globaltable $Localtable $FilestoCheck $blockinfo
$level %tabletype $ptr.names %ptr.names $parentpacking $packing %TYPE

```
$typevarflag >==> (21.a)<ArrayType $GlobalTable $LocalTable $filestocheck
$blockinfo $slevel $tabletype $typevarflag $ptrnamesdef $parentpacking $localtable
$packing $typevarflag ==>(21.b)<RecordType $globaltable $slocaltable
$filestocheck $blockinfo $slevel $tabletype ==> (21.c)<RecordType $globaltype
$parentpacking $packing $typevarflag $tabletype $ptrnames ==>(21.d) <FileType
$ptrnamesdef $parentpacking $filestocheck $slocaltable $tabletype $typevarflag>
$globaltable $localtable $blockinfo $slevel $tabletype> $typevarflag
$ptrnames $ptrnamesdef $parentpacking $packing $type> $typevarflag
==> (21.e) <Settype $globaltable $filestocheck $blockinfo $slevel
$tabletype $ptrnames $parentpacking $packing $type
$typevarflag>

<ArrayType $globaltable $localtable $filestocheck $blockinfo $slevel
$ptrnames $ptrnamesdef $parentpacking $typevarflag>
$tabletype $ptrnames $' <idxtype $globaltable $filestocheck $blockindo $slevel
==> ARRAY [' <idxtype $globaltable $filestocheck $arrayactualtype
$tabletype $ptrnames $packing $arrayactualtype
$fileselect $typevarflag >

<idxtype $globaltable $filestocheck $blockinfo $slevel
$tabletype $ptrnames $packing $array(packing/indextype/
$comptype] $fileselect $typevarflag> ==> <indextype $globaltable
$localtable $filestocheck $blockinfo $slevel $tabletypeindex $indextype>
<idxcomptype $globaltable $filestocheck $blockinfo
$slevel $tabletype $ptrnamesdef $packing $comptype $fileselect
$typevarflag>

<idxcomptype $globaltable $tabletypeindex $filestocheck $blockinfo
$slevel $tabletype ==>' JNOR <componenttype $spacking $comptype $fileselect
$typevarflag> $Jnor <componenttype $globaltable $tabletypeindex
$filestocheck $blockinfo $slevel $tabletype $ptrnames $ptrnamesdef
$packing $comptype:fileselect $typevarflag>|=" <Herecomptype==
$actualtype:fileselect $packing> <idxtype $globaltable $tabletypeindex
$filestocheck $blockinfo $slevel $tabletype $ptrnamesdef
$packing $actualtype $fileselect $typevarflag>

<IndexType $globaltable $localtable $filestocheck $blockinfo $slevel
$tabletypeindex $type.actualtype> ==> <Simpleidtype $globaltable
$localtable $filestocheck $blockinfo $slevel $tabletypeindex $unpacked
```

```
$type $1) $typevarflag>  <where type.actualtype.typeacttag=ordinal>

<Componenttype $globaltable $tabletypeindex $filestocheck $blockinfo
$level $tabletype $ptrnames $ptrnamesdef $packing $comptype $comptype.
$fileselect $typevarflag> ==> <typecbenoter $globaltable $tabletypeindex
$filestocheck $blockinfo $level $tabletype $ptrnames $ptrnamesdef
$packing $comptype $typevarflag>

<Recordtype $globaltable $localtable $filestocheck $blockinfo $level
$tabletype $ptrnames $ptrnamesdef $parentpacking $packing *(record
(packing;$layout;$recordtableout;$tmptr) $fileselect;$parentpacking)
$typevarflag> ==> *RECORD* <Fldust $globaltable $localtable $filestocheck
$blockinfo $level $tabletype $ptrnames $ptrnamesdef $packing $layout
$botptr $1) *recordtableout $typevarflag $fileselect>

<Fldust $globaltable $localtable $filestocheck $blockinfo $level
$localtable $ptrnames $ptrnamesdef $packing *() $nil $recordtablein
$recordtablein $fixpart $taglist $typevarflag $false> ==> NIL

<Fldust $globaltable $localtable $filestocheck $blockinfo $level
$tabletype $ptrnames $ptrnamesdef $packing $fixpart $recordtablein
$recordtableout $localtable $ptrnames $ptrnamesdef $packing $layout $recordinfo
==> (c)<Componart $ptrnames $tabletype $blockinfo $botmptr $recordinfo
$recordtableout $recordtableout $fixpart $taglist $fileselect
$typevarflag> ==> (1)<variantpart $tabletype $ptrnames $fileselect
$blockinfo $level $recordinfo $recordtablein $recordtablein $filestocheck
$botmptr $recordtableout $fixpart $taglist $layout
$fileselect $typevarflag> i<

<Compart $globaltable $localtable $filestocheck $blockinfo $level
$tabletype $ptrnames $ptrnamesdef $packing $layout $layout $botmptr
$recordtablein *recordtableout $fixpart $taglist $filestocheck
$typevarflag> ==> <Recordsection $globaltable $localtable $filestocheck
$blockinfo $level $tabletype $ptrnames $ptrnamesdef $packing $layin
$layout $botmptr $recirdtablein $recordtablein $recordtablein $taglist
$fileselect $typevarflag> i<
<Recordsection $globaltable $localtable $filestocheck $blockinfo
```

```
slevel $tabletype1 $ptrnames $ptrnamessection spacking slayin
slayoutsection $btuptr $recordtableto $recordtablesection $tixpart
staglist $filcselecti> <Restrecsection $globaltable $tabletype1
$tilestocheck $plockinto slevel $tabletype $ptrnamessection $ptrnamesdef
spacking slayoutsection $layout $btuptr $recordtablesection $ptrnamesdef
$recordtableout $tixpart starclist $fileselect2 $typevarflag>
         <Here fileselecti or tileselect2>

<Restrecsection $globaltable $tabletype1 $filestocheck $plockinfo
slevel $tabletype $ptrnamessection $ptrnamesdef spacking slayoutsection
$layout $btuptr $recordtablesection $recordtableout $tixpart staglist
$fileselect2 $typevarflag>

                 ==> <variantpart $globaltable $tabletype1
$filestocheck $plockinfo slevel $tabletype $ptrnamessection $ptrnames
spacking slayoutsection $layout $btuptr $recordtablesection
$recordtableout $globaltable $tabletype1 $filestocheck $plockinfo slevel
<Complpart $globaltable $tabletype $ptrnamessection $plockinfo slevel
$tabletype $btuptr $recordtablesection $recordtableout $tixpart staglist
$layout $btuptr $recordtablesection spacking slayoutsection $tixpart staglist
$fileselect2 $typevarflag>
         <Restrecsection $globaltable $tabletype1 $filestocheck $plockinfo
slevel $tabletype1 $ptrnamessection $ptrnamessection spacking
slayoutsection $layoutsection $btuptr $recordtablesection
$recordtablesection $tixpart staglist $false $typevarflag>    ==> NUL

<variantpart $globaltable $tabletype $filestocheck $plockinfo slevel
$tabletype $ptrnames $ptrnamesdef spacking $address $tagtypeactual?
variantaccesslist] $recprotablein $recordtableout $taglist slayin
slayout $fileselect $typevarflag>
                ==> "CASE" <variantselector
$globaltable/tabletype1 (plockinfo) $filestocheck slevel $spacking
$recordtablein $recordtablevariant slayin slayout $discrimination
$tagtypeactual $taglistelement $tixpart $typevarflag> <variantlist
$globaltable $tabletype1 $taglistelement $filestocheck $plockinfo slevel $tabletype
$ptrnames $ptrnamesdef spacking $plockinfo slevel $recordtablein
$recordtableout $taglist $taglistelement $tagtypeactual $[]
$taglistvalue $fileselect $typevarflag> <CheckforCompletenessofCaseLabels
$tagtypeactual $taglistvalue)

<variantlist $globaltable $tabletype $filestocheck $plockinfo slevel
```

8.a)

$tabletype $ptrnames $ptrnamesdef $packing $variantaccesslistin $variantaccesslist
$recordtablein $recordtableout $taglist $taglistelement $tagtypeactual
$taglistvaluein $taglistvalue $fileselect $typevarflag

$globaltable $tabletype1 $filestocheck $blockinfo $level $tabletype $ptrnames
$ptrnamesdef $packing $variantaccesslistin $variantaccesslist $recordtablein
$recordtableout $taglist $taglistelement $tagtypeactual $taglistvaluein
$taglistvalue $fileselect1 $typevarflag> ==> <Variant

<Variantlist $globaltable $tabletype1 $filestocheck $blockinfo $level $tabletype $ptrnames
$ptrnamesdef $packing $variantaccesslistin $variantaccesslist $recordtablein
$recordtableout $taglist $taglistelement $tagtypeactual $taglistvaluein
$taglistvalue $fileselect1 v $fileselect2> ==> <Variant $globaltable $tabletype1
$filestocheck $blockinfo $level $tabletype2 $ptrnames $packing
$variantaccesslistin $variantaccesslist $recordtablein $recordtableout $taglist
$taglistelement $tagtypeactual $taglistvaluein $tabletype1 $fileselect1
$typevarflag> <Restvariant $globaltable $ptrnames $ptrnamesdef $packing $blockinfo $level
$level $tabletype $ptrnames $recordtableout $taglist $taglistelement
$variantaccesslist $recordtableactual $taglistvalue1 $taglistvalue $fileselect2 $typevarflag>
$tagtypeactual $taglistvalue1

<Restvariant $globaltable $tabletype $filestocheck $blockinfo $level $tabletype $ptrnames
$ptrnames $packing $variantaccesslistin $variantaccesslist $recordtable
$recordtable $staglist $staglist $taglistelement $recordtable $recordtable
$false $typevarflag> ==> nul

<Restvariant $globaltable $tabletypein $filestocheck $blockinfo $level $tabletypeout
$ptrnames $ptrnamesdef $packing $variantaccesslistin $variantaccesslist
$recordtablein $recordtableout $taglist $taglistelement $tagtypeactual
$taglistvalue1 $fileselect $typevarflag> ==> <Variantlist $globaltable
$tabletypein $filestocheck $blockinfo $level $tabletypeout $ptrnames $ptrnamesdef
$packing $variantaccesslistin $variantaccesslist $recordtablein $recordtableout
$taglist $taglistelement $tagtypeactual $taglistvalue2
$fileselect1 $typevarflag>

<Recordsection $globaltable $localtable $filestocheck $blockinfo $level $tabletype $ptrnames
$ptrnamesdef $packing $layin $layout $btmptr $recordtablein $recordtableout
$fixpart $taglist $type $fileselect $typevarflag>

```
==> <identlist $recordtablein $namelist > ';'
    $level <typedenoter $globaltable $tilestocheck $blockinfo
    $level $tabletype $ptrnames $ptrnamesdef $packing $type $car(identlist)
    <typevarflag>
    <makelayoutandrecordtable $layin $layout $namelist $recordtablein
    $recordtableout $style $fixpart $packing $staglist $level $taytypeactual >

.1)                  <makelayoutandrecordtable $layin $layout s[] $recordtablein
    $recordtableout $level $typeactual $fixpart $staglist>        ==> NUL

.1.a  <makelayoutandrecordtable $layin $layout $name ^ namelist $recordtablein $recordtable &
      {name ==> (field(taglist);type;level)} $typeactual $false $staglist $staglist
      slevel>        ==> <makelayoutandRecordtable $layin $layout $namelist $namelist
      $recordtableout $typeactual $true $staglist $level>
      $recordtablein $recordtable $type $true $staglist $level>

.1.c  <makelayoutandrecordtable $layin $layout $name^namelist $recordtablein $recordtable|
      {name ==> (field(taglist);type;level)} $typeactual $true $staglist $1
      ^{name}        ==> (field(taglist);type;level)} $typeactual $true $staglist $1
      ==> <makelayoutandrecordtable $layin $layout $name1 $namelist $recordtablein
      $recordtableout $type $true $staglist $level>

.a    <variantselector $symboltable $tilestocheck $level $packing $recordtablein $recordtablein
      ^{name} ==> (field(taglist);type;level)} $layin $layout $discriminated
      $type.typeactual $fixpart $staglist $typevarflag>
      ==> ident ^{name} (";" ident ^name) <Here mode=Env(name2)};layin $sttypevarflag>
      <Here mode.kind = type and name1 does not belong to filestocheck >
      <Here type=node.type><here type.typeactual.ord =char _enum and
      type.typeactual.range=
      subrange V d1) & <makeudyout .$layin $layout $fixpart $(union(discriminat;
      type)f{type.typeactual int & subrange>

.b    <variantselector $symboltable $tilestocheck $level $packing $recordtablein $recordtablein
      $layin $layout $freeunion $type.typeactual $[] $fixpart $staglist
      $typevarflag>
      ==> "IDENT" $name <HereMode = mode(enu[name])>
      <heremode.kind = type> <HereTypeActual = node.type> <here type;
      typeactual.typeactualtag = ordinal & type.typeactual.range.rangeselect
```

b.1 <MakeLayout $layin = $uprange> <MakeLayout $layin $layer $typeactual>
     <MakeLayout $layin $layer $typeactual>   ==> NUL

b.2 <MakeLayout $layin $layin^typeactual $true $typeactual>   ==> NUL

b.? <Variant $globaltable $localtable $tilestocheck $blockinfo $level $tabletype $ptrnames
     $otrnamesdef $packing $variantaccesslistin $variantaccesslist $recordtablein
     $recordtableout $staglist $taglistelement $stagtypeactual $staglistvaluein
     $taglistvalueout $fileselect $typevartidq>   ==> <CaseConstantList $globaltable
     /localtable(blockinfo) $tactypeactual $taglistin $taglistin $taglistvalueout
     $caseconstlist> <MakeTagList $taglist $taglistelement $caseconstlist
     $taglistout > "(" <Tlist $globaltable $ptrnames $caseconstlist
                        $tabletype $ptrnamesdef $packing $layout
     $recordtablein
     $btmptr $recordtablein
     $recordtableout $false $taglistout #tileselect > ")" <MakeVariantAccessList
     $variantaccesslistin $variantaccesslist $caseconstlist $btmptr>

1.a <MakeTagList $taglist $[] $caseconstlist $taglist>   ==> NUL

1.b <MakeTagList $taglist $(name) $caseconstlist $taglist^(name) $caseconstlist>   ==> NUL

2   <MakeVariantAccessList $variantaccesslistin $variantaccesslistin^(caseconst,btmptr)
     $caseconst^caseconstlist $caseconst $btmptr>   ==> <MakeVariantAccessListin1
     $variantaccesslistin $variantaccesslistin $caseconstlist
     $btmptr>

2.a <MakeVariantAccessList $variantaccessin $variantaccess s[] $btmptr>   ==> NUL

     <CaseConstList $symboltable $tactypeactual $taglistvaluein $taglistvalueout

```
<caseconstantlist> ::= <CaseConstant $symboltable $tagtypeactual %constvalue
   <here taglistvalue1> taglistvalue1 = taglistvalue1 constvalue1 & caseconstlist1
   1constvalue1> scaseconstantlist1 {,<CaseConstant $symboltable
   $tagtypeactual %constvalue2>/> $taglistvalue1 = taglistvalue1 constvalue2
   $caseconstlist1 = caseconstlist1 constvalue2 <here taglistvalueout =
   taglistvalue1 & caseconstlist1 = caseconstlist1>

<CaseConstant $symboltable $tagtypeactual $typeactual.range.valued>   ==> <Constant $symboltable
   $typeactual> <Polaxegordinaltype=Equivalence $tagtypeactual $typeactual
   $mismatch = false> <here $typeactual.range.valued BELONGSTO
   $tagtypeactual.range>

<Settype $globaltable $localtable $filestocheck $blockinfo $level $table type
   $parentpacking $spacking $(sattelementtype);false;parentpacking>   ==>   "SET OF"
   <Basetype $globaltable $localtable $filestocheck $blockinfo $level %tabletype
   $elementtype>

<Basetype $globaltable $localtable $filestocheck $blockinfo $level $tabletype $packing
   $type>   ==> <Simpletype $globaltable $localtable $filestocheck $blockinfo $level
   $tabletype $spacking $type>[ = false] <where type.typeactual.typeactualtag
   =ordinal>

<filetype $globaltable $localtable $filestocheck $blockinfo $level $tabletype $ptrnames
   $ptrnamesdef $parentpacking $spacking &tile(filedomain);void);true;parentpacking
   $typevarflag>   ==>   "FILE OF"  <Componenttype $globaltable $localtable $filestocheck
   $blockinfo $level $tabletype $ptrnames $ptrnamesdef $packing %filedomain
   $fileselect1 $typevarflag> <here fileselect1 = false>

<Pointertype $symboltable $localtable $parentpacking %(pointer(address);false;parentpacking)
   $ptrnames $ptrnamesdef $typenamesdef $typevartag %localtableout
   $level>   ==> <Domaintype $symboltable $localtable $ptrnames %ptrnamesdef
   address $typevartag $localtable>
```

<Domaintype $symboltable $localtable $ptrnames $ptrnamesdef %address $typevarflag
   $localtableout $level> ==> ident #name <Here flag:=name BELONGSTO symboltable
   /localtable(dummy %lockinfo) flag?
      true: <address := address(name)> <where
            address.mode.kind.kindtag = type>
            <localtableout := localtable>
            <typevarflag?
               true: <localtableout := localtable
                     @(name ==> (type;void;level))>
                     <Ptrnamesdef := ptrnames*name>
                     <Here address := address(name)>
               false: <error>

<Typeid $symboltable $logmode.type> ==> ident #name <Here mode=mode(env(name))>
   <where mode.kind = type>

a   <Var-Dec-Part $globaltable $localtable $localtabletype $tilestocheck $lockinfo $lockcheck
      $filestocheck> ==> nil

b   <Var-Dec-Part $globaltable $localtable $localtabletype $tilestocheck $lockinfo $lockcheck
      $remfilestocheck> ==> "VAR" <Here localtable1=localtabletype and remfiles1
      =filestocheck> $localtable1<var-Dec $globaltable $localtable $localtable1
      $remfiles1 $lockinfo $level %localtable $remfiles2;> %localtable1
      $localtable2 $remfiles1=remfiles1 $remfiles2 <Here localtablevar1=localtable1
      $remfilestocheck= remfiles1><where remfilestocheck= [j]>

<Vardec $globaltable $localtable $tilestocheck $lockinfo; $level %localtablevar $remfilestocheck
   $filestocheck> ==> <Identlist $localtable %namelist> <Typedenoter $globaltable $localtable
   $tilestocheck $lockinfo $level %tabletype $ptrnames> $ptrnamesdef $unpacked %type>
   <Here mode=(var;type;level)> <outidentlistinlocalenviron $localtable %localtablevar
   $namelist %mode>

1.a <Outidentlist $symboltable $localtable %localtable $l] $mode>   ==> NUL

```
1.b   <PutIdentListInTypBoolTable %localtable & (name -->) mode} $name^namelist $mode>
      ==> <PutIdentList...%localtable $localtable $namelist $mode>


<Proc-Fn-Dec-Pt $globaltable $localtable $globallabels %labelstojump
        $localtableproc $level $reftablein $reftableout $blockinfo>
      ==> <Here-localtablein>%reftablein
          $reftablein1 = reftablein1 = reftablein=localtablein>
          $reftableinl ,<Procdec/Fncdec $globaltable %localtablein
          $globallabels %labelstojumpout %localtableout $localtableout $level
          $reftableinl %rcitableproc $blockinfo> %localtableout=localtablein=localtableout
          %labeltojumpj=labeltojumpl U labelstojumpout $reftableinl = reftableproc
          <Here localtapleproc=localtablein & labelstojump=labelstojumpl>
          & reftableoutl = reftablein1>


<Proc-Dec  $globaltable $localtable $globallabels %labelstojump $localtableout $level
          $reftablein $reftableout $blockinfo>
          ==> <Proc-Heading $globaltable $localtable $localtable
          $level+1 $proc-table $name $reftablein $reftableini $localtablehead
          $formaltag $blockinfo $plan $forwardtag>*<Directive $localtablehead $proctable
          $forwardflagsname $plan $localtable $blockinfo $level $void>
<Here localtableout= localtablehead & (name -->)(proc(plan)/(void/false;unpacked)/level*
          <Proc-block $globaltable/localtableout(blockinfo) $proctable $level+1
          $reftableinl $reftableout $(name;procblock) $globallabels $labelstojump>


<Proc-Heading $globaltable $localtable $level $proctable $name $plan $forwardflag
          $reftablein $reftableout>
          ==> "PROCEDURE" ident $name <Forwardentry $localtable $nam
      $presenceflag $mode $forwardtag $blockinfo $localtableout>
      %Selection
        presenceflag : true
          <Here mode;kind=forwardproc> <Here plan=mode;kind;plan & proctable=
          mode;kind;environ & reftableout = reftablein & forwardflag = true>
          presenceflag = false
      ?SelectionFormaltag
        false :<Here reftableout=reftablein U (name-->)[]][];false;level>
        true: <reftableout=reftablein & forwardflag = false>
          <Here plan=[] & proctable = {}>,<Formalarlist $globaltable/localtable $nam
          (blockinfo) $level $formaltag $plan $proctable>
```

<ForwardEntry $localtable $name $presenceflag $mode
$localtableout>
    ==> ?if v[name]
        true : <Here presenceflag = true & mode = mode(env[name])>
                <Here localtableout=localtablein & localtableout=(name==>mode)>
        false : <Here presenceflag=false & localtableout =
                localtablein>

<FormalParaList $symboltable $plan $proctable $level $formaltag >
    ==> (<FormalParaSection $symboltable $formaltag $forwardflag >
          $formaltag
          $plan1 $proctable1) <FormalParaSection $symboltable $plan1 $plan2 $proctable1
          $proctable2 $level $formaltag | $plan1 $( ) $proctable1 $level
          <Here plan=plan1 & proctable=proctable1>)"

<FormalParaSection $symboltable $plan1 $planin $proctablein $proctableout $level $formaltag>
    ==> <value-Para-spec $symboltable $plan1in $planin $proctablein $proctableout $plen
        $formaltag> | <var-Para-spec $symboltable $planin $proctablein $proctableout $lev
        $proctableout $level $formaltag> | <Proc-Para-Spec $symboltable $planin
        $planout $proctablein $proctableout $level $formaltag> | <Func-Para-Spec
        $symboltable $planin $planout $proctablein $proctableout $level $formaltag>

<Value-Para-Spec $symboltable $planin $planout $proctablein $proctableout $level
        $formaltag>
    ==> <Identlist $proctablein $namelist> ":" <Typedid $symboltable
        $name $type>
        ?SelectionformalEag>
        false : <PutidentListinLocalTable $proctablein $proctableout $namelist
                $(formalval;type;level) >
                <Here $proctableout = proctablein>
        true : <UpdatePlan $planin $planout $namelist
                $formalpar(simplepar(void;assignment;void;
                (formalval;type;level);0)) >

0.1.a) <UpdatePlan $planin $planin sl 1 $typeactual > ==> lambda

0.1.b) <UpdatePlan $planin $planin1 *lanin1 * typeactual $name * namelist $typeactual >
    ==>

```
<VarParSpec <UpdateplanShelf $planin %planin1 $namelist $typeactual >
        $symboltable $planin %planout $proctableout $level
        $formaltag >
                ==> 'VAR' <Trentlist $proctablein %namelist > ';'
        <TypeIdConformantSchema $symboltable $proctablein $proctable1 %type $level
        <Here node = (formalvar;type;level)) >
        formaltag ?
                false : <putident;istinlocaltable $proctable11 $proctableout
                        $namelist $node>
                true : <UpdatePlan $planin $planout $identlist
                        $(;void;typeequivalence;void;node;0)
                        <Here proctablein = proctablein >

.1)  <TypeIdConformantSchema $symboltable $proctablein $proctableout %type $level
        $formaltag >
                ==>(a)
        <Typeid $symboltable %name %type><Where type.fileselect =false>
        <Here proctableout = proctablein>
                ==>(51-1;b)
        <ConformantArraySchema $symboltable %type $level $formaltag >

<ConformantArraySchema $symboltable $proctablein $proctableout
        %(typeactual;false;unpacked) $level $formaltag $proctableout
        'ARRAY' '[' <IdxTypeSpec $symboltable $proctablein %proctablein $proctableout
                                %typeactual >

.1)  <IdxTypeSpec $symboltable $proctablein $proctableout %(confarray(indextype;elementtype))
        $level $formaltag >
                ==>
        <IndexTypeSpec $symboltable $proctablein %indextype $level
        $formaltag> <IndexType2Spec $symboltable $proctable
                        $proctableout.elementtype $elementtype $level $formaltag >

.2)  <IndexType2Spec $symboltable $proctable $proctableout %elementtype $level
        $formaltag >
                ==>(a) 'I' 'OF' <TypeIdConformantSchema $symboltable $proctableout %elementtype $level
                        $proctableout %elementtype
```

&level $formaltag >

.2b) <IndexType2Spec $symboltable $proctable $proctableout
     $actualtype;false;unpacked) &level $formaltag >
     ==> .<idxtypespec $symboltable $proctable $proctableout %actualtype
            &level $formaltag >

<IndexTypeSpec $symboltable $proctable $proctableout %actualtype &level $formaltag >
     ==> ident $name? ... (ident $name2 ?; <OrdinalTypeId $symboltable
                                              %actualtype>
     formaltag ?
          false : <here mode = (bound1?;(actualtype;false;void;
                                unpacked);level)><here proctableout =
          proctable ε {name? --> {name2 -->mode } >
          true : <here proctableout = proctablein > >

<OrdinalTypeId $symboltable %type.actualtype >
     ==><TypeId $symboltable %name %type >
     <here type.actualtype.actualtype = ordinal >

<ProctableSpec $symboltable $proctablein &level $simplepar(void;typeequivalence;void;Mode;0)
     $proctablein $proctableout &level $formaltag>
     ==>
     <Procheading $symboltable $proctablein &level $proctablein $name $plan
          $forwardflag ; $reftableout $true > <here proctableout = proctablein >
          <here mode = (formalproc(plan);(void;false;unpacked);level)>
          forraltag ?
               true : <here proctableout = proctablein >
               false : <here proctableout= proctablein ε {name -->mode} >

<FuncParSpec $symboltable $pl000 splanin = formalpar(simplepar(void;typeequivalence
     ;void;mode;0) $proctablein $proctableout &level $formaltag > ==>
     <Funcheading $symboltable $proctablein &level $proctable $name $plan $forwardflag
          $proctableout $true $proctableout &resulttype >
     <here mode = (formalfunc(plan);(resulttype;false;unpacked);level) >
     formaltag ?
          false: <here proctableout = proctablein ε {name --> mode } >
          true : <here proctableout = proctablein >

<Directive $localtable $proctable $name $plan $localtableout $forwardflag $blockinfo
$level $resulttype >

  <Where forwardflag = false > ident $name < where $name = forward>
  blockinfo.pktype ?
    procblk : localtableout = localtable & (name -->)(forwardproc(plan;
              $proctable);(void;false;unpacked);level } >
          funcblk:<here localtableout = localtable & (name -->)
              (forwardfunc(plan);proctable);(resulttype;false;void;unpacked);
              level } } >

<ProcBlock $globaltable $proctable $level $reftablein &reftableout $blockinfo
    $globallabels %labelstojump >
          ==> <Block $globaltable $proctable
    %labelstojump si] $blockinfo $level $reftablein &reftableout >

<FunctionDec $globaltable $localtable $globaltable $level $labelstojump $localtableout $level
    $reftablein $reftableout $blockinfo >
          ==>

  <FuncHeading $globaltable/$localtable $level $functable $name $plan
        $forwardflag $localtablehead $formaltag $resulttype $blockinfo > ',"
  (59.a) <Directive $localtable $forwardflag $blockinfo $level $resulttype $localtableout
        $forwardflag $localtablehead $functable $plan $name $resulttype >
  (59.b) <Here localtableout = localtable $blockinfo $level $resulttype $localtableout
        (resulttype;false;unpacked);level) >
    <Funcblock $globaltable/$localtable(blockinfo) $functable & { name -->)
        (localfunc(plan)(resulttype;false;unpacked)(level+1) } $(name;funcblock)
        $globallabels $labelstojump $reftablein $reftableout >

<FuncHeading $globaltable $localtable $level $functable $name $plan $forwardflag
    $localtableout $formaltag $resulttype $blockinfo> ==>
  "FUNCTION" ident $name <Forwardentry $localtable $name $presntflag $mode
                                    $localtableout >
  presentflag ?
    TRUE: <here mode.kind = forwardfunc > <Here plan = mode.kind &
        functable = mode.kind.environ & forwardflag = true >
    FALSE: <Here plan =[ ] & functable =[ ] &forwardflag=false>

```
l <for albarlist $slobaltable/localtableout(blockintr) &olan
  $functable $levpl $fcrualtag > 1 ;
  <ResltType $s/solproLric &resulttype >

<ResultType $symboltable &type.typeactual >

  <Typeid $symboltable &name & type> <where type.typeactual.typeactualtag =
    ordinari real / pointer >                              ==>

<FunctionBlock $symboltable $functable $level splockinfo $globaltable &labelstojump
  &reftableout >                                          ==>

  <Block $symboltable $functable $globallabels &labelstojump s( )
    $blockinfo $level &reftablein &reftableout>
```

CS-1901- M-KAC-SYN